EFFICIENT SHARED CACHE MANAGEMENT FOR MULTICORE PROCESSORS

A Thesis
Presented to
The Academic Faculty

by

Yuejian Xie

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
College of Computing

Georgia Institute of Technology
August 2011

EFFICIENT SHARED CACHE MANAGEMENT FOR MULTICORE PROCESSORS

Approved by:

Gabriel H. Loh, Committee Chair
College of Computing
Georgia Institute of Technology

Gabriel H. Loh, Advisor
College of Computing
Georgia Institute of Technology

Milos Prvulovic
College of Computing
Georgia Institute of Technology

Hyesoon Kim
College of Computing
Georgia Institute of Technology

Hsien-Hsin S. Lee
School of Electrical and Computer Engineering
Georgia Institute of Technology

Sudhakar Yalamanchili
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: 18 May 2011

ACKNOWLEDGEMENTS

Funding and equipment were provided by a grant from Intel Corporation, and funding was provided from the National Science Foundation under Grant No. 0702275.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

In modern multi-core processors, various resources (such as memory bandwidth and caches) are designed to be shared by concurrently running threads. Though it is good to be able to run multiple programs on a single chip at the same time, sometimes the contention of these shared resources can create problems for system performance. Naive hard-partitioning between threads can result in low resource utilization. This research shows that simple and effective approaches to dynamically manage the shared cache can be achieved. The contributions of this work are the following: (1) a technique for dynamic on-line classification of application memory access behaviors to predict the usefulness of cache partitioning, and a simple shared-cache management approach based on the classification; (2) a cache pseudo-partitioning technique that manipulates insertion and promotion policies; (3) a scalable algorithm to quickly decide per-core cache allocations; (4) pseudo-LRU cache partition approximation; (5) a dynamic shared cache compression technique that considers different thread behaviors.

CHAPTER I

INTRODUCTION

Thesis statement: Shared caches in multicore processors need to and can be managed to improve overall system performance by adding low overhead hardware.

1.1 Shared Resource in Multicore Processors

The well known Moore's law indicates that the number of transistors we can put on a chip doubles every two years [39]. In recent years, the chip manufacturers found it is hard to continuously scale single thread application performance, so they switched the microprocessor philosophy from frequency to parallelism. After the switch, we had entered a "multicore" and "multithread" era. Multicore processors are also known as Chip Multi Processors (CMP) [43], which indicates that multiple cores are all integrated on a single chip. Intel's multicore products include Core, Core 2 and Nehalem architecture; AMD also has successful products such as Athlon and Phenom. Apart from the desktop and server area, the embedded processor company ARM also designs multicore processors such as Cortex-A9 and ARM11MPCore.

There are a lot of benefits of putting multiple cores on a single chip. For example, a multicore processor can run multi-threaded applications faster than a multiprocessor system that consists of several single core processors, because the processors can communicate faster as the distances between them are shorter. It is also relatively cheap to have a dual core multicore processor than two coupled single core processors. By having a multicore processor, the end user can run multiple independent programs at the same time, such as watching a movie and editing a document.

In many implementations of multicore processors, the last level cache (LLC) is shared

by all of the cores on the chip. It is also common that the memory bus is also shared. The shared resource design is helpful when there is imbalance of resource demand between different cores. For example, if every core has its own private cache, one core cannot use another core's cache space even when other cores do not need all of their space at all. In a shared cache design, one core can potentially take the entire cache space when it is very active, and this will result in higher resource utilization. However, if the programs have aggressive behaviors and compete with each other, and the resource is left unmanaged, the efficiency can drop significantly. It is even possible that the performance could be worse than executing two programs one by one serially on a single core processor. It is a challenge to design such a structure which enables the sharing between programs but also avoids competing and thrashing problems.

While a multicore processor has several physical cores integrated, the "Simultaneous MultiThreading" processor (SMT) uses another type of technique for multiprocessing [55, 56]. The SMT processor generally only has one set of core pipeline, but in a given time, instructions from different threads can be simultaneously executed in different stages of the pipeline. This multiprocessing ability is achieved relatively less expensively because there is no great changes to the basic architecture. It needs to enlarge some structure size such as the register file to hold information from all the threads.

In an SMT processor, a lot more resources are shared than in a multicore processor, such as the level 1 data cache, branch predictor, prefetcher, instruction queue, etc. If two concurrently running programs share these structure without significant contention, it would be more efficient to use an SMT processor than a multicore processor. For example, one program has a lot of integer computation and the other is always doing the floating point computation so they share the execution units without problems. However, if two programs do not have compatible behaviors, it is also possible that they compete with each other and eventually both get slowed down.

Some processors are a mixture of both a multicore and an SMT processor. For example, the Intel Core i7-960 processor has four physical cores and each core can support two simultaneous threads. In total it can run eight threads at the same time. All these threads

are sharing a single 8MB level 3 cache. In this case, it is a big challenge for how to efficiently manage all the shared resources among eight threads.

1.2 Related Work

Shared resource management is a general problem in many fields in computer science. For example, the CPU is usually shared by all the concurrently running processes. The operating system task scheduler needs to coordinate all the processes to use the shared resource: the CPU time. In a network switch or a router, packets can come from different ports and go to different ports. It needs to know how to handle those packets via the channel, which is also a shared resource. In the case of multicore and multithread processors, we are most interested in the on-chip shared resource between all the running processes and threads. In the past few years, researchers had performed a good amount of study on management of shared resources such as memory bus bandwidth and last level cache. This section will review several related works.

In multicore processors, the memory system resources are not controlled. It can result in destructive interference between threads. Nesbit et al. proposed fair queuing memory systems [42]. The Fair Queuing memory scheduler borrows the ideas from computer networking to optimize for a Quality of Service objective function. In a fair queuing memory system, different threads are ensured to receive the allocated fraction of the memory system, regardless how much memory load other threads place on the memory system.

Instead of designing a specific scheduling policy for the memory controller, Ipek et al. proposed a technique that uses principles of reinforcement learning (RL) [24]. The RL approach can overcome the difficulty that conventional fixed, rigid access scheduling policy cannot learn and optimize the long-term performance. The self-learning approach is also good at adapting to dynamic workload behavior.

The last level cache of a CMP processor is usually designed to be shared. Researchers investigate the interference on the shared cache between running threads. It comes with the problem of the unfair cache occupancy. Kim and Solihin proposed fair cache sharing concepts to avoid thread starvation and priority inversion [30]. The paper proposed several

cache fairness metrics that can measure the cache sharing fairness degree. Two of them are shown to be strongly related with execution-time fairness. Based on these metrics, a dynamic partitioning algorithm is designed to manage the shared L2 cache. It is reported that the fairness and the system throughput are both improved while applying the fair cache techniques.

Qureshi and Patt proposed Utility-based Cache Partitioning (UCP) [45]. The paper observed that one can build hardware called utility monitor to dynamically measure the different cache space needs from different programs when they are running. After the utilities from all the programs are learned, an accurate cache partition decision can be made so that the system is optimized to have better throughput. The paper also proposed techniques to use an approximate algorithm to make the techniques scalable to more cores and larger associativity.

Manipulating cache replacement policy is another category of managing the shared cache. Qureshi et al. found that changing the insertion policy of the LRU replacement policy to inserting new cache lines to the LRU position instead of the MRU position can help to reduce the lifetime of dead cache lines. The Dynamic Insertion Policy (DIP) combines both LRU and MRU insertion policy and dynamically switches between them, and leads to significant performance speedup [44].

Jaleel et al. further extended DIP policy to consider multicore environment. In the Thread Aware Dynamic Insertion Policy (TADIP) technique [27], different threads are observed separately so their insertion decisions are also made differently. It is reported to be performance effective and scalable to many cores.

The SMT processor has the problem of deciding which thread to fetch instructions from. The default round-robin fetch policy may not work well if different threads behave significantly differently. Tullsen et al. investigated a set of fetching policies for SMT processors [55]. It is shown that the ICOUNT solution is the most effective approach to solve the instruction clogging problem. This solution gives priority to the thread that has the fewest instructions in decode, rename and the instruction queues. ICOUNT was reported to be able to provide significant throughput gain for eight-thread workloads.

Researchers also found the learning based approach also works with the SMT resource management problems. Choi and Yeung proposed a learning-based SMT processor resource distribution via Hill-Climbing [12]. This approach can dynamically learn the impact that resource distribution decisions have on performance. This information is fed back to the distribution system to decide a better allocation in the future. The hill-climbing algorithm can greatly reduce the learning time so that overall the system can reach the best distribution quickly. The experiments show that Hill-Climbing performs better than the ICOUNT approach.

Alexander et al. described that the Branch History Table (BHT) can be split between two threads, so that every thread gets its prediction independently [3]. A dynamic algorithm can be used to switch the BHT and the counter cache in either unified mode or split mode. Ramsay et al. evaluated different design choices of the BHT and the history register [47]. Each can be either shared or split. The experiments include both Gshare predictor and the YAGS predictor. The results showed the system performance is not significantly impacted by the branch predictor sharing because the thread level parallelism can hide the branch misprediction latency.

1.3 Our proposal

This dissertation aims to propose mechanisms to improve system performance by managing the shared resource, specifically for the last level cache. There are a few challenges for this. First, the machnisms must be able to significantly improve the system performance. Second, they need to be practical to use. For example, one can statically study two programs' behaviors and carefully design a mechanism to allocate resources specifically to these two particular programs; it works but it is not practical to use. Ideally, the mechanism should be able to dynamically capture the programs' behavior and manage the resource on-the-fly. There should be no need to statically pre-analyze the behavior. Third, if we build additional hardware to help improve system performance, the hardware must be very cheap to build. If the additional hardware itself takes too much area and power, one can just simply enlarge the shared resource budget so the competing problem probably just goes away.

The dissertation includes a variety of works about managing the shared last level cache. Chapter 2 first introduces an application classification technique which can dynamically predict the usefulness of cache management scheme. The following chapters propose several low overhead cache management schemes that can significantly improve the overall system performance. Chapter 7 investigates a new aspect of cache management: how to manage a shared last level cache that uses data compression.

CHAPTER II

DYNAMICALLY IDENTIFY THE PROBLEM

The simultaneous execution of multiple programs on a multicore processor with a shared on-chip cache can result in cache interference that reduces system throughput, overall fairness and quality of service. Such situations may occur due to the presence of a program with a large memory footprint with frequent accesses, but low reuse characteristics which end up evicting a large number of cache lines used by other cores.

This chapter is mainly to demonstrate one part of the thesis statement: Shared caches in multicore processors **need** to be managed to improve system performance. The classification technique proposed in this chapter is used to explain the typical cache competition scenario which results in overall system performance slowdown. As a result, we need cache management schemes to improve the system performance.

Recent research efforts have reported on a large variety of static and dynamic cache management schemes to partition cache resources among the multiple cores [9, 28, 30, 45, 50, 52]. However, cache management schemes may not provide consistent speedup over an unmanaged cache, because for some cases the cache competition problem does not exist or has little impact. Moreto et al. analyzes program characteristics to explain why some multiprogrammed workloads benefit from cache partitioning while other workloads do not [40]. Using a few simple metrics, they show that they can accurately classify benchmarks to predict when dynamic cache partitioning would be of value. Other studies have also proposed similar-in-spirit classification schemes for workload creation purposes [35, 45]. These classification techniques can be employed in a post mortem fashion to explain speedup phenomena after the fact. However this chapter presents a new, simple classification technique that can be employed in vivo to dynamically monitor the instantaneous behavior of applications. On-the-fly classification of workload cache behaviors has many potential applications.

For example, the classification information can be used to improve cache partitioning algorithms, or the information can be fed back to the operating system so that programs with conflicting/ incompatible behaviors can be rescheduled to nonoverlapping timeslots, or even to different chips (in a multisocket system). Chapter 5 proposes a simple and low overhead cache management scheme based on our dynamic classificiation technique.

2.1 Existing Classification Schemes

As briefly mentioned in the introduction, several other works have presented a variety of approaches to classify programs' cache behaviors in a multi-core context. This section presents a brief overview of some of these.

2.1.1 Lin's "colors"

Lin et al.'s study aimed to duplicate past work on cache partitioning by using OS-level page coloring to directly allocate L2 cache resources between two programs on a real system (as opposed to in simulation) [35]. As part of this work, they classified programs into one of four classes to which they assigned colors. To perform this classification, they consider the performance degradation observed when running a program using only a 1MB L2 cache compared to the baseline configuration with 4MB. Any program with greater than 20% slowdown was classified as Red and greater than 5% slowdown (but less than 20%) as Yellow. Out of the remaining programs with less than or equal to 5% slowdown, the program is classified as Green if the total number of L2 accesses is greater than or equal to 14 misses per thousand cycles, otherwise the program is in the Black category.

While Lin et al.'s color-based classification scheme may be useful for workload creation, it cannot be easily used for dynamic, on-the-fly classification of program behavior. In particular, computing the performance slowdown would require simultaneously running two copies of the program on two cores, each with their own dedicated L2 caches. One cache then further needs to be crippled to only use one fourth of the cache capacity, and then finally performance between the two would be compared. To perform dynamic classification, the two copies would have to be synchronized to execute the exact same code, which means that the version with the full 4MB core will constantly be waiting for the slower 1MB

version to catch up. If a separate core was available, then it would make more sense to simply schedule two programs on separate cores rather than co-scheduling them together and then use two additional cores to determine their respective stand-alone performances. Overall, these constraints render Lin et al.'s classification approach completely impractical for in vivo cache-sharing behavior classification. To be fair, Lin et al.'s approach was not designed for dynamic scenarios; we make note that it is not trivial to simply "port" their approach, which motivates the need for a new scheme.

2.1.2 Moreto et al.'s Dual Metrics

Moreto et al. introduced two metrics to classify programs into three categories, which they use to explain when speedups can be obtained by cache partitioning [40]. The first metric, $w_{P\%}$, is the number of ways needed by a program to obtain at least $P\%$ of its maximum IPC (i.e., the IPC achieved if the program had sole usage of all n ways of the L2 cache). The second metric, $w_{LRU}(th_i)$, is the (average) number of ways used by each thread when running simultaneously (assuming LRU replacement). Moreto et al.'s results show that they can accurately predict when cache partitioning will be beneficial, but similar to Lin et al.'s scheme, the metrics cannot be easily monitored in an online fashion (i.e., $w_{P\%}$ requires comparison against the performance when using the full L2 cache, which in turn requires a complete redundant execution of the program).

2.1.3 Utility-based Classification

Qureshi and Patt studied the marginal utility of increasing set-associativity on programs [45]. In particular, they describe applications as belonging to one of three classes. High-utility programs exhibit continued performance improvements (or cache miss rate reductions) as the number of ways are increased. Low-utility programs do not gain much benefit from allocating more cache ways. Saturating-utility programs show performance improvements with more cache ways, but only up to a point. Past this point, the allocation of additional ways does not significantly increase performance any further. Their categorization appears to be largely based on visual analysis of the per-program utility curves; they do not provide any formal, algorithmic means of classifying programs into these classes and as a result, it

is not obvious how one would go about implementing a dynamic (in hardware) mechanism for classifying programs based on their criteria. Their application analysis also includes CPI rates as the cache associativity is varied; if this information is required for classification, then it would be impractical to dynamically collect this information as it would require running the n concurrent copies of the program, each with a different number of ways allocated.

Qureshi and Patt also used a simple classification scheme for the purposes of generating interesting multi-core workloads [45]. In particular, for k concurrently running programs (sharing the L2 cache), the weighted speedup is equal to $\sum_{i=1}^{k} IPC_i/SingleIPC_i$, where IPC_i is the committed instructions-per-cycle (IPC) rate of program i when concurrently running with all other $k-1$ programs; $SingleIPC_i$ is the committed instructions-per-cycle rate when program i is running alone with full usage of the entire L2 cache. This is slightly different from Lin et al.'s classification as this groups the workloads rather than the individual benchmarks. Since this weighted-speedup-based classification requires comparing the performance of programs run separately and together, it is also nearly impossible to use as in an online classification mechanism.

2.1.4 Chandra et al.'s Miss Models

Chandra et al. proposed three detailed models for predicting the impact of cache sharing among multiple threads [8]. The models provide an estimate of the number of additional L2 misses caused by sharing when compared to running a thread by itself. In particular, their Inductive Probability model predicts the number of such misses with an average error of only 4%. Unfortunately, the model is fairly involved; the large number of complex statistical computations would be very difficult to directly implement in hardware. While the model can in theory be used to classify programs' behaviors in the context of L2 cache sharing, the granularity of the model output is much finer than is necessary for our purposes.

2.2 An "Animalistic" Taxonomy

In our proposed scheme, we classify benchmarks into intuitive "animal" personalities based on a few simple heuristic metrics. The main difference between our approach and the

previously proposed schemes is that all of our metrics can be directly derived at runtime in hardware.

2.2.1 Our Four Animal Types

Turtles: There are some applications that simply do not make much use of the shared last-level (L2) cache. This may be because the program simply has very few memory instructions to begin with, or it may be that the program has a very small working set that completely fits within the level-one caches and therefore rarely accesses the L2 cache. We call these programs turtles as they are small and slow moving (with respect to frequency of L2 accesses).

Sheep: In a cache that employs dynamic cache partitioning, some applications simply are not sensitive to the number of ways allocated to them. These applications may actually exhibit a high rate of L2 accesses, but even with an allocation of only a few ways, these programs can achieve a low L2 miss rate. We call these programs sheep as they are gentle and tame, and not easily perturbed by other applications.

Rabbits: Some applications are very sensitive about the number of ways allocated to them. Such applications access the L2 cache fairly frequently, but if provided with a sufficient number of ways, the overall miss rate can be kept low. The performance can rapidly degrade if there is an insufficient cache allocation. We call these programs rabbits as they are fast moving (with respect to frequency of L2 accesses), delicate (in that they are more easily impacted by other programs) and tend to be happier with more space to run around in.

Tasmanian Devils: Our last class of applications simply do not play well with others. These applications access the L2 cache very frequently, but still have very high miss rates. As a result, such applications do not derive much benefit from occupying the cache (in terms of hit-rate reduction), and furthermore they tend to negatively impact other applications since the frequently missing accesses tend to kick out cache lines that are useful to other programs. We call such programs Tasmanian Devils, or just devils for short.

2.2.2 Dynamic Classification Algorithm

As discussed before, other previously proposed classification schemes require multiple configuration performance comparisons (e.g., speedup compared to the situation where the program runs alone with full ownership of the L2) and therefore are not amenable to dynamic, on-chip classification mechanisms. Our animal-based taxonomy has been designed to allow for on-the-fly monitoring of programs' animal classes. We first review the Utility Monitor concept (which our classifier uses) and then describe the exact criteria used for determining a program's animal type as well as implementation issues.

The Utility-based Cache Partitioning work by Qureshi and Patt proposes a simple mechanism for dynamically tracking the benefit or utility of allocating additional cache ways to a program, while keeping the overhead under control through set sampling [45]. This Utility Monitor (UMON) consists of $n+1$ counters per core for an n-way set associative cache. If a core actually had all n ways allocated for its sole use, a cache hit in the i^{th} position in the LRU stack causes the i^{th} UMON counter to be incremented, and the $n+1^{st}$ counter tracks misses. What each counter ends up representing is the number of additional hits that could be achieved if one more way were to be allocated to this program, and so the counters are sometimes called marginal gain counters. In concept, the UMON implementation requires that the L2 cache keep one set of shadow tags per core, but the set sampling technique greatly reduces this overhead by implementing shadow tags for only a subset of the cache sets and assuming that the behavior observed for these sampled sets are representative. Qureshi and Patt demonstrate that the difference between set sampling and implementing full sets of shadow tags is quite small. In the results presented section, we simply use full shadow tags for implementation simplicity, with the understanding that set sampling can be employed to greatly reduce the monitoring overhead.

We use a combination of dynamically collected information to determine the "animal" personalities of programs. We make use of the following metrics:

- *Accesses*: The total number of accesses to the L2 cache by the program. This includes instruction, data and prefetch requests.

- $Misses_{solo}$: The total number of L2 misses if the program had sole use of the entire n ways of the cache.
- $MissRate_{solo}$: The relative L2 miss rate if the program had sole use of the entire n ways of the cache ($Misses_{solo}/Accesses$).
- $WaysNeeded_{k\%}$: The smallest number of ways needed to achieve a miss rate that is greater than or equal to $k\%$ of $MissRate_{solo}$.

The *Accesses* metric can be easily tracked by a single integer counter per core. The $Misses_{solo}$ metric requires checking accesses against the shadow tags that track the simulated cache contents as if the program had sole access to the L2 cache. Apart from an integer counter to track the number of misses, this metric does not require any additional overhead since we are already paying the price of shadow tags for the utility monitor. At first glance, computing $MissRate_{solo}$ and $WaysNeeded_k$ seem to require expensive division operations that would make dynamic tracking of these metrics impractical. We will now describe the final classification criteria and then return to explain why division operations are not needed.

Algorithm 1 Pseudo-code for our Animal-based Classification Rules.

```
if Accesses < 1000 then
    Animal = Turtle
else if ((MissRate_solo > 10%) OR (Misses_solo > 4000)) then
    Animal = Devil
else if (WaysNeeded_95% > n/2) /*n = set − assoc */ then
    Animal = Rabbit
else
    Animal = Sheep
end if
```

Based on the metrics defined, we classify programs based on the heuristic rules described in Algorithm 1. The intuition for these rules follows the qualitative descriptions of the "animals" given at the start of this section. If the application does not access the cache a lot, then it will be a turtle. If the application does access the cache a bit more frequently, then we examine its miss rate and its total misses. If the miss rate is high or the total (absolute) number of misses is extremely high, then we classify the program as a devil. The first criteria captures the situation where the program brings in a significant number of cache lines, but does not have a high reuse rate and so caching these lines is not very useful.

The second criteria captures the situation where the program simply misses in the cache very frequently. In this case, regardless of whether the program has a high overall L2 hit rate, the sheer number of accesses will rapidly cause cachelines owned by other programs to get flushed out from the cache (i.e., they will rapidly be demoted in the global LRU stack and then evicted). In the remaining two cases, if the program needs a medium (or more) number of ways to attain a low number of misses (compared to $Misses_{solo}$), then we classify it as a rabbit. Finally, the sheep access the cache with some regularity, but even a few ways are sufficient to maintain decent hit rates.

The exact constants employed in our rules were chosen through empirical analysis. We derived these thresholds by conducting some initial experiments of the UCP algorithm. Workloads that have different UCP speedups are grouped together first. We then plotted figures of the four metrics for all the benchmarks. By correlating the metrics and the UCP speedup, we are able to identify effective thresholds for our classification. Note that these constant thresholds are dependent on a particular platform (cache size, associativity, memory latency, etc.). For example, for a smaller cache, the *Accesses* threshold also needs to be decreased because one program can occupy a significant portion of the cache with fewer number of accesses. For a cache that has larger memory parallelism (larger MSHRs, more banks, etc.), the $Misses$ and $MissRate$ thresholds could also become larger. Note that even though the thresholds need to be set for different platforms, for a particular design, the process of determining the thresholds only needs to be performed once. We also found that the results and trends did not vary by any significant amount if we slightly change the thresholds. Returning to the issue of implementation complexity, we first consider computing the condition $MissRate_{solo} > 10\%$. This is equivalent to computing $Misses_{solo}/Accesses > 10\%$, which can be rewritten as $Misses_{solo} > Accesses \times 0.1$. By choosing a slightly different threshold, say 12.5%, the multiplication by 0.1 can be replaced by a multiplication by 0.125 (same as division by eight) which can be trivially implemented as a right-shift by three positions. Next, for $WaysNeeded_{95\%}$, computing 95% of the hit rate would also involve an unattractive multiplication by 9 followed by division by 10. By choosing a threshold such as 93.75%, which equals $1 - 1/16$, we can compute $WaysNeeded_{93.75\%}$ with a rightshift

and a subtraction. For a practical implementation, several of the other constants could be replaced by the closest power-of-two, such as replacing 1000 by 1024 which simplifies testing for less-than to a NOR-operation of any bits of the counter left of the tenth least-significant bit. Since we found that the overall trends did not vary significantly with slightly different values of these parameters, we simply did not optimize them for implementability in this work. The main point of this discussion is to convince the reader that our classification scheme can be easily tweaked to make implementation easy.

The parameters were chosen for a sampling frequency of one million cycles. That is, we collect the metrics over an interval of one million cycles, and at the end of the interval, we make the decision of what type of an animal each program is acting like. We then reset all relevant counters and then re-evaluate program behavior after another one million cycles. If shorter or longer sampling intervals are desired, then the constants involved in the classification rules will need to be scaled appropriately. Similarly, with set sampling, the constants will also need adjustments proportional to the degree of sampling.

Note that during different phases, programs may exhibit different animalistic traits. We have observed that some programs act like devils during some phases, and then later may be more sheep-like. Similarly, programs (or program phases) may exhibit different degrees of animal traits. For example, one devilish program may satisfy the $Misses_{solo} > 4,000$ criteria by having 4001 such misses, which another program may have $Misses_{solo} = 200,000$. We informally refer to the former as a "small" devil, and the latter as a "large" devil. If a program consistently exhibits devilish behavior throughout its entire execution, then we also call it a "pure" devil. Similarly, due to the somewhat arbitrary criteria of $WaysNeeded_{95\%} > n/2$, there exists a continuum of behaviors between very "sheepish" (i.e., needing only a single way in the L2 cache) to very "rabbity" (i.e., needing all n ways to achieve a high hit rate).

2.3 Predicting Usefulness of Cache Partitioning

In our first case study, we demonstrate how to use our classification scheme to predict when cache partitioning will be useful. In particular, we use Lin et al.'s color-based classification

Table 1: Baseline Processor Configuration

Fetch Width	16 bytes per cycle
Decode Width	insts(4-1-1-1 μops)
Function Units	3 IALU, 1 IMul, 1 FAdd, 1 Div, 1FMul, 1 Load, 1STA, 1STD
ROB Size	96 entries
RS Size	32 entries
LDQ/STQ Size	32/20 entries
Commit Width	4 μops per cycle
IL1/DL1	32KB, 8-way, 64-byte lines
Shared L2	4MB, 16-way, 64-byte lines
Main Memory	SDRAM, 800MHz bus(DDR), 9-9-9

as a point of comparison [35]. In particular, their results showed that cache partitioning provides the greatest benefits when a Red program is paired/co-scheduled with a Green program, and that the speedups are relatively small for all remaining color combinations. After first briefly explaining our simulation methodology, we will enumerate our benchmarks and show how they are classified under both Lin et al.'s and our own schemes. We will then show how Qureshi and Patt's Utility-based Cache Partitioning (UCP) performs for different workloads and demonstrate that our classification is more accurate at predicting when cache partitioning will be fruitful.

2.3.1 Simulation Methodology

For our simulations, we used the pre-release version of the SimpleScalar toolset for the x86 ISA [4], and we extended it to perform cycle-level modeling of a multi-core processor. The simulator accounts for the contention for cache/memory buses, finite MSHR capacity, a memory controller with request reordering [48], and DDR2 SDRAM timing. The simulated processor configuration is based on the Intel Core 2 Duo; the full details are listed in Table 1. The current version of the x86 SimpleScalar toolset does not support multi-threaded programs, and as a result we simply make use of multi-programmed workloads. In particular, we use benchmarks from SPEC2006 from both the integer and floating point suites. We use reference inputs, and benchmarks with multiple inputs are distinguished by additional numerical suffixes (e.g., soplex.1, soplex.2). Due to incomplete system call

emulation support in the x86-version of SimpleScalar, some applications currently cannot be simulated. Figure 1 lists the applications and their baseline statistics. We use SimPoint 3.2 to select representative samples of each benchmark [23]. Prior to detailed cycle-level simulation, we warm the caches for 250 million instructions per application in a round-robin fashion. We then run the timing simulation for 500 million instructions per benchmark. We follow the methodology used by Qureshi and Patt where once a benchmark reaches its simulation instruction limit, we freeze the statistics for that application, but then continue simulating it so that it continues to compete for cache resources (otherwise, the other program would suddenly have access to the entire L2 cache).

2.3.2 Classification

Figure 1 lists each benchmark along with relevant statistics. The applications are sorted by Lin et al.'s color-based classification scheme. For the Red and Yellow classes, the programs are sorted by the performance slowdown (between running with 4MB and 1MB L2 caches), and the Green and Black applications are sorted by the L2 access rates (listed per 1000 instructions). These are the primary metrics used in their classification approach. For each benchmark, we also include a distribution of the time spent in each animalclass based on our proposed scheme. For example, bzip2.4 spends 24% of the time acting like a sheep, 65% of the time as a rabbit, and 12% of the time as a devil. The overall classification is based on which state the program spends the majority of its time in. There are no clear correlations between the two classification approaches; e.g., not all devils are in the red group and vice-versa.

2.3.3 Predicting Partitioning Speedups

We now compare the performance of Qureshi and Patt's Utility-based Cache Partitioning (UCP) algorithm over a range of workloads.

We first present results for workloads formed based on Lin et al.'s color-based classification scheme. For each combination of colors (excluding Black which has so few L2 accesses to have any real impact on partitioning), we randomly formed several workloads (we evaluate many more workloads than, for example, Lin et al.'s study). In this work, we

Benchmark Name	Base IPC	4MB/1MB Slowdown	L2 Access Rate (PKI)	% Time Spent as:				Overall Animal	Color Class
				Turtle	Sheep	Rabbit	Devil		
bzip2.4	1.35	48.8%	37.9	0.0%	23.6%	64.8%	11.7%	Rabbit	RED
soplex.1	0.34	34.1%	35.5	0.5%	50.1%	42.1%	7.3%	Sheep	RED
bzip2.1	1.62	31.9%	22.1	0.0%	39.8%	58.6%	1.6%	Rabbit	RED
mcf	0.16	29.8%	48.4	0.0%	32.5%	0.2%	67.3%	Devil	RED
omnetpp	0.54	24.1%	27.0	0.0%	0.8%	6.9%	92.4%	Devil	RED
bzip2.5	1.22	22.8%	16.9	0.5%	50.1%	42.1%	7.3%	Sheep	RED
bzip2.6	0.97	22.3%	19.4	1.2%	78.0%	20.4%	0.4%	Sheep	RED
bzip2.2	1.13	21.9%	16.9	0.0%	73.9%	25.9%	0.2%	Sheep	RED
bzip2.3	1.12	21.3%	20.3	0.0%	88.8%	11.2%	0.0%	Sheep	RED
hmmer.1	1.02	20.4%	11.4	0.0%	37.6%	61.1%	1.2%	Rabbit	RED
h264ref.1	1.09	16.3%	10.7	2.2%	46.1%	51.7%	0.0%	Rabbit	YELLOW
perl.3	1.08	15.3%	12.9	0.0%	11.9%	48.7%	39.4%	Rabbit	YELLOW
astar.1	1.15	10.1%	16.4	0.0%	66.3%	30.0%	3.7%	Sheep	YELLOW
h264ref.2	0.81	10.0%	10.0	19.7%	79.3%	1.0%	0.0%	Sheep	YELLOW
hmmer.2	1.01	8.7%	10.0	0.0%	99.4%	0.0%	0.6%	Sheep	YELLOW
astar.2	1.16	8.2%	9.8	0.0%	39.4%	60.6%	0.0%	Rabbit	YELLOW
h264ref.3	0.78	6.9%	14.5	18.5%	81.5%	0.0%	0.0%	Sheep	YELLOW
leslie3d	0.48	6.3%	27.8	0.0%	8.3%	0.0%	91.7%	Devil	YELLOW
cactusADM	0.81	5.7%	25.9	0.0%	52.7%	0.0%	52.7%	Sheep	YELLOW
soplex.2	0.31	5.1%	27.7	0.0%	0.0%	0.0%	100.0%	Devil	YELLOW
libquantum	0.42	0.0%	50.3	0.0%	0.0%	0.0%	100.0%	Devil	GREEN
go.1	0.78	2.8%	31.0	0.2%	99.4%	0.0%	0.5%	Sheep	GREEN
sphinx3	0.49	3.1%	30.0	0.0%	0.3%	0.0%	99.7%	Devil	GREEN
go.4	0.80	2.9%	28.2	0.0%	99.5%	0.3%	0.2%	Sheep	GREEN
gromacs	0.92	2.0%	27.7	0.0%	95.6%	0.4%	4.1%	Sheep	GREEN
perl.2	1.33	0.1%	27.5	0.0%	100.0%	0.0%	0.0%	Sheep	GREEN
go.2	0.83	2.7%	26.8	2.5%	96.8%	0.2%	0.5%	Sheep	GREEN
go.5	0.88	1.3%	25.3	0.9%	98.9%	0.2%	0.0%	Sheep	GREEN
milc	0.33	-0.2%	24.7	0.0%	0.0%	0.0%	100.0%	Devil	GREEN
go.3	0.69	2.0%	23.6	1.4%	80.2%	0.1%	18.3%	Sheep	GREEN
zeusmp	0.88	4.0%	21.1	0.0%	66.1%	0.0%	33.9%	Sheep	GREEN
lbm	0.22	0.0%	20.2	0.0%	0.0%	0.0%	100.0%	Devil	GREEN
gcc.2	0.63	0.7%	17.8	0.0%	5.3%	0.0%	94.7%	Devil	GREEN
gcc.3	1.27	0.1%	13.5	0.0%	100.0%	0.0%	0.0%	Sheep	BLACK
dealII	0.80	1.2%	11.0	46.1%	14.6%	1.1%	38.1%	Turtle	BLACK
perl.1	1.31	2.7%	9.1	46.6%	25.5%	2.4%	25.5%	Turtle	BLACK
gcc.1	1.52	-0.4%	8.7	0.0%	55.9%	0.0%	44.1%	Sheep	BLACK
sjeng	0.78	0.2%	3.9	0.0%	84.2%	0.0%	15.8%	Sheep	BLACK
namd	1.39	1.7%	1.5	88.6%	5.3%	0.0%	6.1%	Turtle	BLACK

Figure 1: Benchmark classification

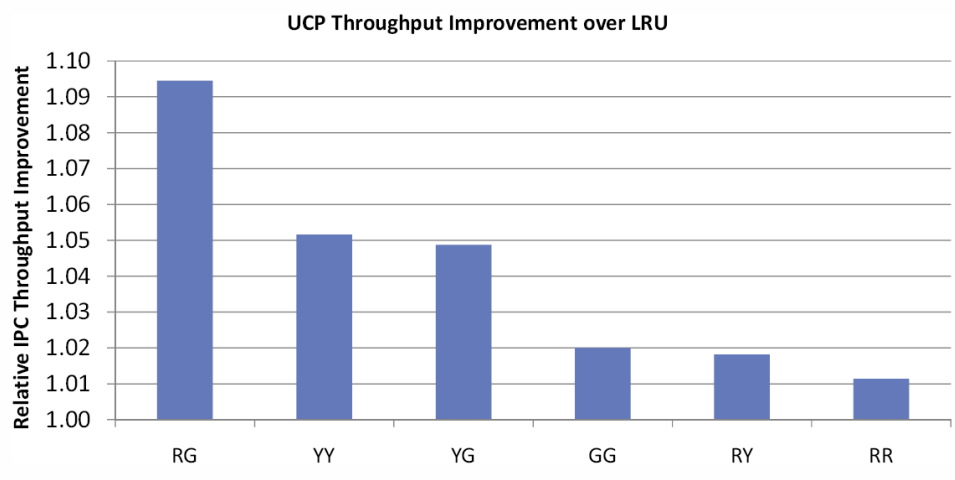

Figure 2: IPC throughput improvements for UCP compared to traditional LRU of different workload groups as determined by Lin et al.'s color-based classification

report IPC throughput (sum of the per-benchmark IPCs); other potential metrics include the weighted speedup and fair speedup, but we only stick to a single metric for brevity. Our results show that workloads combining Red with Green programs (RG) result in the highest performance improvements for UCP; this is in agreement with that reported by Lin et al. Figure 3 shows the per-workload breakdown for UCP on the RG grouping. From these results, however, we see that the performance improvements are not consistent across the different application pairings. The Yellow-Yellow (YY) grouping also has a smaller overall performance improvement according to Figure 2, but the per-workload results in Figure 4 show that here, too, the improvements are in-consistent. After presenting results based on our animalclassification scheme, we will return to these inconsistencies and explain them with our approach.

We then took all of these workloads, and regrouped them based on our animal classification scheme. Figure 5 shows the average throughput improvements across each set of animal pairings. The rabbit-devil combination yields the greatest improvements for UCP followed by devil-sheep. Figure 6 shows the individual throughput rates for each of the rabbit-devil workloads. So not only does the overall group average show strong performance gains, but the gains are quite consistent across all of the individual workloads as well. This makes sense, as the rabbits are all sensitive to how much effective cache capacity gets allocated to

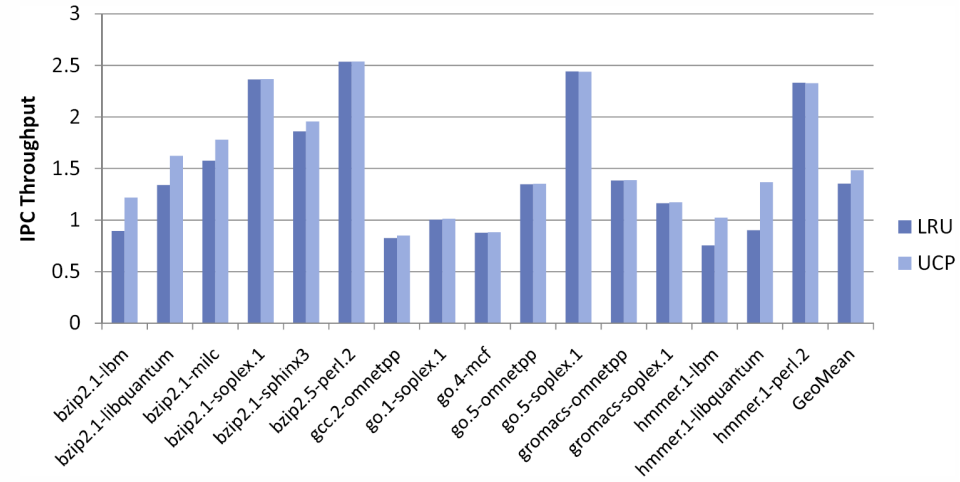

Figure 3: IPC throughput for the Red-Green workloads

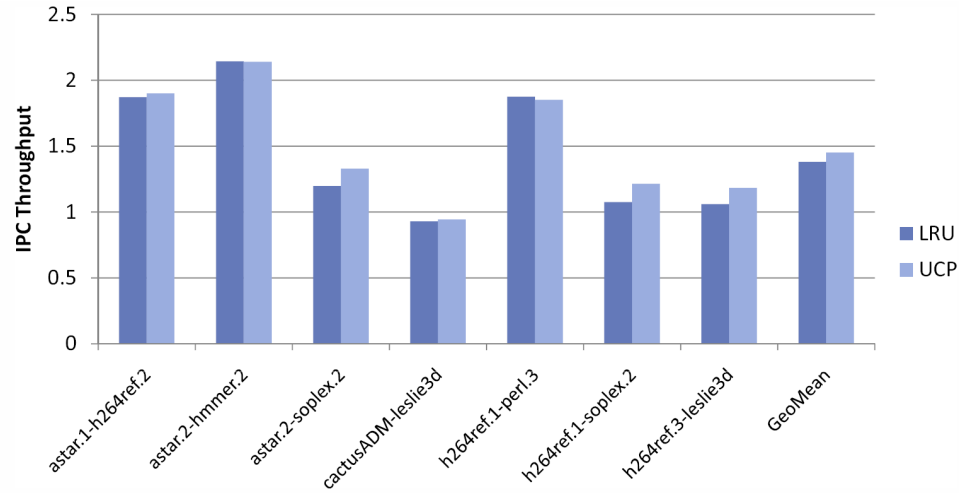

Figure 4: IPC throughput for the Yellow-Yellow workloads

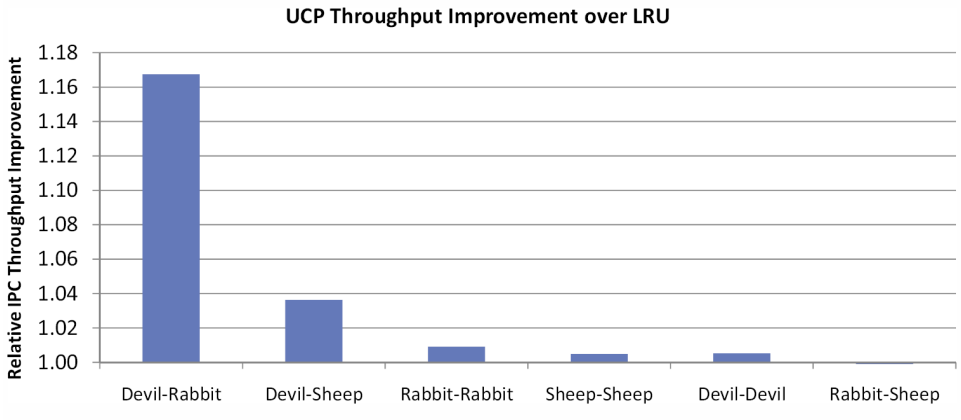

Figure 5: IPC throughput improvements for UCP compared to traditional LRU of different workload groups as determined by our animal-based classification

them, while the devils tend to thrash the cache. Applying dynamic cache partitioning in this scenario prevents the rabbit applications from getting hurt by the devils. The results for the devil-sheep workloads are similar (Figure 7), except that the magnitude of benefit is reduced. This is also consistent with the fact that sheep tend to need many fewer ways in the cache.

For the remaining workloads, UCP generally provides very little benefit. This makes perfect sense for any of the workloads containing sheep (or turtles), as these applications are largely insensitive to how the other co-scheduled application makes use of the cache. For the devil-devil case, neither application makes good use of the cache, and so both will simply continue to miss in the L2. For the rabbit-rabbit, both programs need a significant number of ways in the cache to maintain relatively low miss rates. It is not to say that UCP does not do a good job at partitioning the cache resources between the applications; UCP by construction does partition the cache very well. The reason why the performance gains are not larger for these workloads is that the natural partitioning that occurs as a result of applying regular old LRU also works very well. This results from the applications having fairly-well matched access rates and miss rates.

Returning to the anomalies observed in the color-based classification, we now use our scheme to explain these exceptions. For the red-green grouping (Figure 3), except for the bzip2.1-sphinx3 and gcc.2-omnetpp pairings, all other workloads that show low (or no)

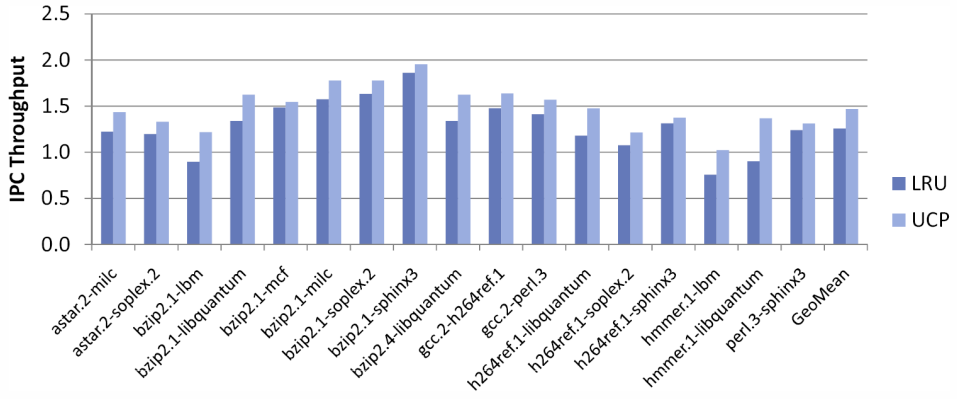

Figure 6: IPC throughput for the Rabbit-Devil workloads

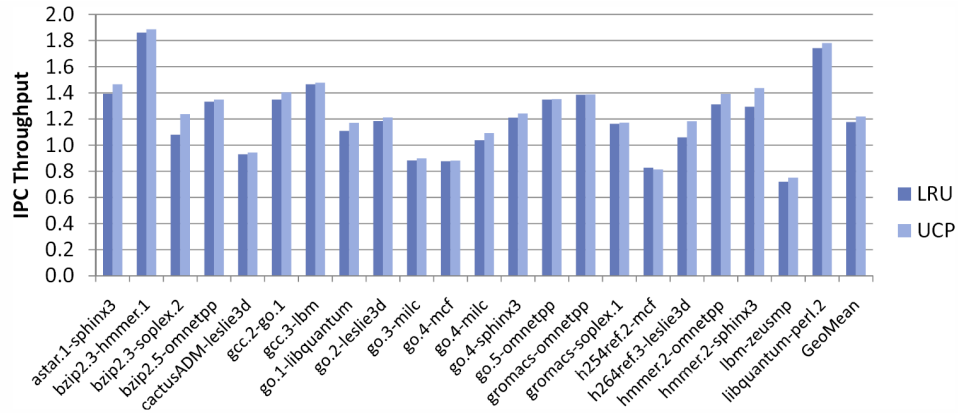

Figure 7: IPC throughput for the Sheep-Devil workloads

speedups involve at least one benchmark that we would classify as sheep. The sheep applications do not have much performance sensitivity to cache allocation, and therefore UCP does not have much impact. We actually classify bzip2.1-sphinx3 as a rabbit-devil pair, in which case we would expect a larger performance gain. The reason that the improvement is relatively modest (+5.1%) is that bzip2.1 is a not a pure rabbit; it only exhibits rabbit behavior for 56% of the time. The last case of gcc.2-omnetpp is a devil-devil pair, for which partitioning also has very limited opportunities to make an impact. For all remaining workloads that did exhibit large performance improvements, our scheme successfully identified these as rabbit-devil pairs.

For the yellow-yellow groupings (Figure 4), the astar.2-soplex.2, h264ref.1-soplex.2 and h264ref.3-leslie3d workloads all show strong gains (+11%, +13% and +12%, respectively)

despite being classified as YY. Our classification scheme groups the first two workloads as rabbit-devil pairings, for which we expect UCP to work well. The last workload is a sheep-devil pairing. For most sheep, an allocation of only 1-2 cache ways is sufficient to maintain high performance, while h264ref.3 is a "fatter" sheep that requires 3-4 ways.

From our results, we believe that our animal classification scheme provides a better means of determining the cache-sharing behaviors of workloads. At least for the workloads evaluated, we have shown that we can more accurately predict when cache partitioning will be beneficial. Our scheme can also be employed as a more meaningful and useful workload creation criteria for future multicore shared cache studies. The fact that our scheme can be measured on-the-fly and provide phasic characterization can also help to better explain observed cache partitioning behavior.

The goal of this work was to introduce our new animal-based classification scheme and demonstrate its usefulness. In particular, we showed that we can use this approach to better identify cache-sharing behaviors and more accurately predict when cache partitioning will be useful. Our classification scheme is able to help us to understand the concret reasons of multicore performance slowdown due to cache competition. And therefore this provides a guideline for the following chapters to further investigate the problem. In Chapter 5, we will show how to use our classification scheme to build a multicore shared-cache management policy that is simpler to implement than UCP while delivering the same (or slightly better) performance benefit. There may be other interesting applications of this animal-based classification scheme. For processors without some form of dynamic cache management/ partitioning, one can provide feedback back to the operating system so that the process scheduler can attempt to avoid co-scheduling incompatible animal types (e.g., use a rabbit-sheep pairing followed by turtle-devil schedule rather than a rabbit-devil followed by turtle-sheep).

CHAPTER III

SCALABLE, OPTIMAL CACHE PARTITIONING VIA DYNAMIC PROGRAMMING

The Utility-based Cache Partitioning needs the decision of how to partition the cache, according to the utility counter information. When the number of the cores and the associativity scale to large number, there will be significant amount of partition possibilities for UCP to evaluate, leading to higher complexity and more power consumption. An approximate algorithm was proposed by Qureshi and Patt to avoid high overhead, but this will result in performance penalty since it could be not an optimal partition decision. This chapter demonstrates the dynamic programming methodology can be used to compute the optimal partition decision with low overhead to improve the overall system performance, so it will support the thesis.

3.1 Description of the Algorithm

For c cores sharing w ways of a cache, there are $O(w^c)$ possible partitionings. The general partitioning problem has been shown to be NP-hard [16]; an optimal polynomial-time algorithm is unlikely to exist unless P=NP. The important factor here is that traditional complexity analysis measures the running time with respect to the *input size*. While our cache partitioning problem does not map exactly to the traditional PARTITION problem studied in complexity theory; the PARTITION problem has an optimal algorithm with polynomial running time with respect to its input value, say n, but the encoding (and hence input size) of n only requires $b = O(\log n)$ bits. Therefore an algorithm with a running time of $O(n)$ (polynomial in n), for example, has a running time of $O(2^b)$ (not polynomial in the input size). Such *pseudo-polynomial* time algorithms are often implementable in practice, however, since the number of bits required to encode input values for real-world problems is typically quite small. Based on this result, we are able to provide an algorithm that performs optimal cache partitioning with polynomial running time in w and c.

Algorithm 2 Pseudo-code for our Optimal cache partitioning algorithm.

```
U, c, w                                              ▷ Utility matrix U, c cores, w ways
procedure OptimalPartition     ▷ This pseudo-code assumes the first element of an array has an index of 1
    for i = 1 to c do                                ▷ Initialize P (temporary matrix)
        for j = 1 to w do
            P[i, j].Hits = U[i, j]
            P[i, j].Part = all j ways assigned to core i
        end for
    end for
    for i = log2(c) downto 1 do                      ▷ Compute U', U'', ...
        P = Compress(P, 2^i)                         ▷ P holds the current U*
    end for
    Return P[1, w].Part                              ▷ Return the final partitioning
end procedure

procedure Compress(P, m)                             ▷ Compress m rows in P down to m/2
    for i = 1 to m/2 do
        P'[i] = Combine(P[2i − 1], P[2i], c/m)
    end for
    Return P'
end procedure

procedure Combine(a,b,s)          ▷ Combine the two rows a and b, each corresponds to s cores
    X[2s] = a[s] + b[s]
    Y[2s] = X[2s]
    for i = 2s + 1 to n do                           ▷ Considering allocation of i ways
        for j = i downto 2s + 1 do
            X[j] = X[j − 1] + a[j − s]               ▷ Partition(j − s, i − j + s)
            if X[j] > Y[i].hits then                 ▷ Track the partition that resulted in the most hits
                Y[i].Hits = X[j]
                Y[i].Part = a[j − s].Part ∪ b[i − j + s].Part
            end if
        end for
        X[2s] = X[2s] + b[i − s]                     ▷ Boundary condition*
        if X[2s] > Y[i].Hits then
            Y[i].Hits = X[2s]
            Y[i].Part = a[s].Part ∪ b[i − s].Part
        end if
    end for
    Return Y
end procedure
```

*This one extra computation here is due to the fact that in each iteration, we need to consider one extra possible partitioning, i.e., the width of each row increases by one compared to the previous, as illustrated in Figure 8(b) and (c).

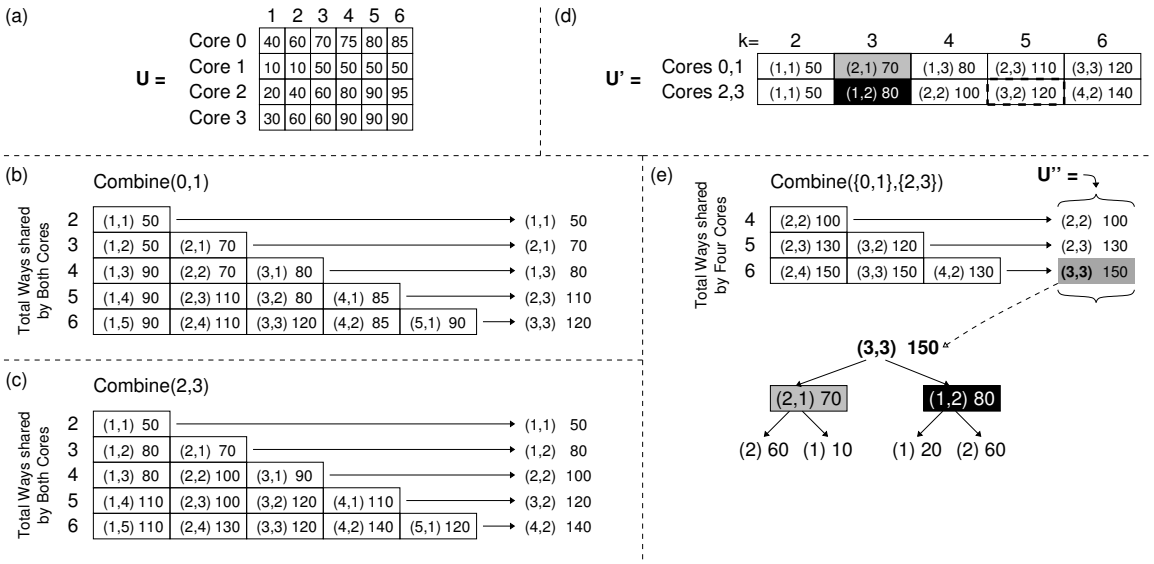

Figure 8: Step-by-step example of our optimal partitioning algorithm for $c = 4$ and $w = 6$

Our algorithm starts with the *Utility Matrix* $\mathbf{U}$, which is simply the list of each of the individual core's UMON hit counters, where each entry $U_i[j]$ is the number of cache hits that core i would have had if given a cache with j ways. The idea is that we will generate a new matrix $\mathbf{U}'$ with half as many rows; the first row will contain $\overrightarrow{U_{0,1}}$ where $U_{0,1}[j]$ is the number of cache hits that cores 0 and 1 would have together if they were sharing a cache with j ways with optimal partitioning. The second row in $\mathbf{U}'$ will contain $\overrightarrow{U_{2,3}}$, and so forth. This results in a matrix with $c/2$ rows and w columns. We repeat this process again to generate $\mathbf{U}''$ where each row now contains the combined utility vectors for four cores at a time, and this process repeats until finally we have a single-rowed matrix $\mathbf{U}^{(\log_2 \mathbf{c})}$. Along with the computation of the $\mathbf{U}$'s, we also need to keep track of the optimal partitionings corresponding to each entry of the $\mathbf{U}$ matrix. Algorithm 2 lists the algorithm in pseudo-code, but it is perhaps easier to understand this algorithm with an example.

3.2 Example of DP algorithm

Consider the case of c=4 and w=6 shown in Figure 8; we present it this way to make it easier to understand how the algorithm works. We start with the $\mathbf{U}$ matrix (a) where each row corresponds to a core, and the entry in column k records the number of hits that that

core would have with a k-way cache (without sharing). The heart of the algorithm is in the COMBINE function, which takes two rows from the utility matrix and computes the optimal partitionings given a number of ways. For example, when we combine the utility information for cores 0 and 1 (b), we start by considering if we only allocated two ways to these two core. Assuming each core must receive at least one way, the only possible allocation is to give each core a single way, denoted by (1,1). Under this allocation, the total number of hits between both cores is 50 ($\overrightarrow{U}_{\text{core 0}}[1]$=40 and $\overrightarrow{U}_{\text{core 1}}[1]$=10). Next, we consider the case where the two cores share three ways, which yields the two possible allocations of (1,2) and (2,1), providing 50 and 70 hits, respectively. For core 1, increasing the number of ways to two does not increase the number of hits (the first two entries in $\overrightarrow{U}_{\text{core 1}}$ are both the same), which is why the number of hits for the allocation (1,2) is identical to that for (1,1). We repeat this procedure for having the two cores share four, five and then finally six ways to fill in all of the entries of the table. For each row, we now search for the partitioning that results in the largest number of hits (the partition with the largest number of hits is shown at the right side of the figure). This now forms the first row of the combined utility matrix $\mathbf{U}'$ to be used in the next level of the algorithm. Figure 8(c) repeats the process for cores 2 and 3; we will not discuss this step as the mechanics are identical to Figure 8(b), but the details are provided for completeness.

Having combined the utilities for cores 0 and 1, and cores 2 and 3, we now have a new matrix $\mathbf{U}'$ with half as many rows as $\mathbf{U}$ (d). Each entry in $\mathbf{U}'$ tells us how many hits would be possible if that *pair* of cores were to be allocated so many ways (and what the partition would be). For example, for cores 2 and 3, if five ways were to be given to them (highlighted with a dashed box around the matrix entry), a total of 120 hits would be possible, and to achieve this, core 2 would be given three ways, and core 3 would get two ways. We now apply the COMBINE function again (e) to the two rows of $\mathbf{U}'$. The single entry in the first row provides the number of hits if four ways were to be shared among all four cores (again assuming a minimum allocation of one way per core). The partition (2,2) means that two ways should be allocated to the first group (cores 0 and 1), and two ways should be allocated to the second group (cores 2 and 3). The second row contains the number of hits assuming a

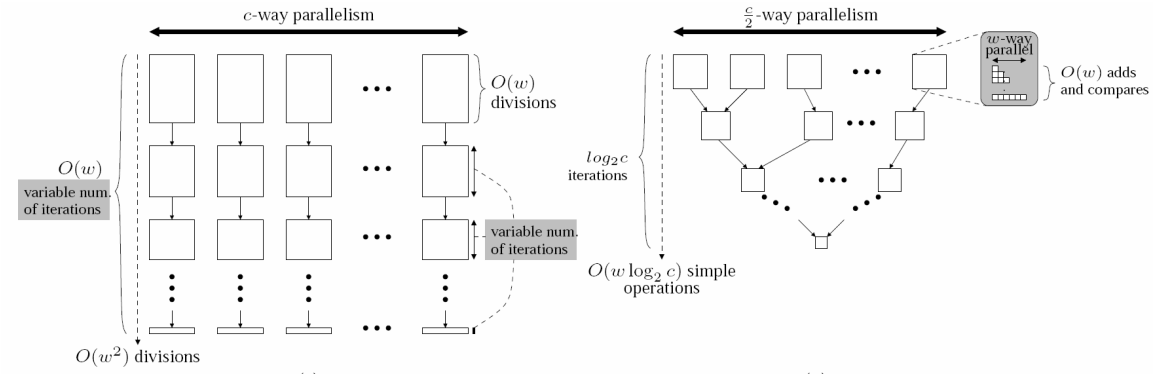

Figure 9: Timing schematic for parallelized implementations of (a) Qureshi and Patt's LookAhead and (b) our OptimalPartition algorithms

total of five ways for the four cores. The total number of hits can be computed in the same way as before, but this time referring to the entries of $\mathbf{U}'$ rather than $\mathbf{U}$. We repeat this for all remaining rows, and then we find the maximum number of hits (and the corresponding partition) per row to form the matrix $\mathbf{U}''$.

At this point, the combined utility matrix $\mathbf{U}''$ only has a single row left (although this is listed vertically in the figure), and so the main part of the algorithm terminates. Given six ways, these four cores will have a total of 150 hits, as shown by the last entry of $\mathbf{U}''$ (shaded). The partition (3,3) indicates that three ways should be allocated to cores 0 and 1, and the other three ways should be allocated to cores 2 and 3. To determine how the three ways should then be further partitioned between cores 0 and 1, we can go back to $\mathbf{U}'$ to determine that for three ways (first row, third column, shaded) the partitioning should be (2,1). Similarly for cores 2 and 3 (second row, third column, in black), the optimal partitioning is (1,2). This yields a final partitioning of (2,1,1,2) for all four cores.

3.3 Complexity Analysis and Implementation

To perform utility-based cache partitioning for $c > 2$ cores, Qureshi and Patt proposed the LookAhead algorithm, which is reproduced in Algorithm 3. LookAhead poses some challenges for direct hardware implementation; in particular, the inner-most loop contains a division operation. If, for example, the divisor is always a power of two, then the division can be replaced by a simple right-shift operation; in LookAhead, however, the divisor is

swept from *balance* to *balance*+*ii*, which requires a fully functional divide unit. The running time of LookAhead will be dominated by the $O(c \cdot w^2)$ division operations. To speed up the algorithm, they propose parallelizing the loop by replicating the partitioning hardware c times across the loop as annotated in Algorithm 3. This now requires c separate divider units to be implemented. Another point of potential complexity is that the number of iterations in the **for** loop in the GetMaxMU function varies depending on *balance*, which in turn depends on the input values for the algorithm (i.e., the hit counters). Similarly, the outer-most **while** loop also has a variable number of iterations depending on how quickly *balance* gets used up. This means that the control logic cannot be hard-wired in advance, but instead the logic must effectively support the evaluation of conditional branches; the hardware to implement the LookAhead algorithm starts to look like a generic instruction-processing pipeline (and recall that we need c copies of this hardware).

Algorithm 3 Qureshi and Patt's LookAhead partitioning algorithm [45].

```
U, c, w                                          ▷ Utility matrix U, c cores, w ways
procedure LookAhead
    balance = w
    allocations[i] = 0 for each competing core i
    while balance > 0 do                         ▷ Input-dependent number of iterations
        for each core i do                       ▷ Parallelize here
            alloc = allocations[i]
            max_mu[i] = GetMaxMU(i, alloc, balance)
            blocks_req[i] = minimum blocks to get max_mu[i] for i
        end for
        winner = application with maximum value of max_mu
        allocations[winner] += block_req[winner]
        balance -= blocks_req[winner]
    end while
    Return allocations
end procedure

procedure GetMaxMU(p, alloc, balance)
    max_me = 0
    for ii = 1 to balance do                     ▷ Input-dependent number of iterations
        mu = GetMUValue(p,alloc,alloc+ii)
        if mu > max_mu then
            max_mu = mu
        end if
        Return max_mu
    end for
end procedure

procedure GetMUValue(p,a,b)
    U = change in misses for core p when the number of
       blocks assigned to it increases from a to b
    Return U/(b-a)                               ▷ Divide operation required
end procedure
```

Our OptimalPartition algorithm has the same $O(c \cdot w^2)$ sequential running time complexity as LookAhead, but we do not require any complex operations like division.

In fact, additions and comparisons are all that we need. We do not have any complex control logic as the few **if** statements are simply for implementing a MAX function which can be directly implemented in hardware as a compare-and-mux operation. The number of iterations in each and every loop of LookAhead can be determined ahead of time; that is, there is no control-flow dependence on the input values. This permits simpler control logic implementations and easier parallelization of the algorithm. Figure 9(a) shows a graphical depiction of the parallel execution of the LookAhead algorithm, with input-dependent loops marked. Each box represents an instance of the GetMaxMU function call, which itself embeds *balance* division operations. Figure 9(b) shows a similar diagram for our OptimalPartition algorithm. Each box represents a call to the Compress function, which includes the calls to the Combine function. Overall, our OptimalPartition has lower hardware overhead (no division, simpler control logic), can be more easily parallelized, and provides an *optimal* partitioning instead of an approximation.

CHAPTER IV

PROMOTION/INSERTION PSEUDO PARTITIONING

To support the thesis that the shared cache in multicore processors can be managed to improve overall system performance, this chapter proposes a new cache management policy called Promotion/Insertion Pseudo-Partitioning (PIPP). Instead of explicitly partitioning the cache by ways, sets or total occupancy, PIPP implicitly partitions (or pseudo-partitions) the cache by simply managing the insertion and promotion policies of the cache. The insertion policy determines where in the eviction priority (e.g., LRU stack) a line should initially be installed [27, 44]. The *promotion policy* determines how the eviction priority should be changed on a cache hit [32]. We show that targeted insertion and promotion can provide effects similar to explicit cache partitioning. Our mechanism's ability to not always insert cache lines at the top of the recency stack also provides benefits similar to adaptive insertion schemes. Furthermore, by not strictly enforcing hard partitions in the cache, our approach allows cores to "steal" cache capacity from other cores, thereby making better use of the total available resources.

4.1 Motivation

The academic and industrial research communities have already made many efforts to manage shared caches in multi-core processors. We now discuss several of the most related prior works so that our contributions can be properly put into context.

4.1.1 Capacity Management

Different programs, or different threads from the same program, executing on a multi-core processor can have different memory capacity requirements (i.e., working set sizes). Given limited cache capacity, the question is how should these resources be divided among the competing cores? One approach is to provide fixed allocations for each core. The amount of space allocated or per-core priority levels can be determined at the software

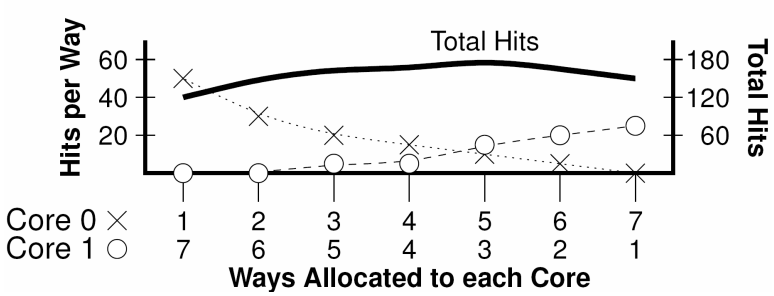

Figure 10: Example utility curves showing the number of additional hits provided for each additional way allocated to an LRU cache, and the total number of hits between the two cores for different partitionings of the cache.

level, for example by the operating system [11, 18, 25, 26, 46, 51], or it could be dynamically determined based on run-time observations of program requirements [9, 31, 45, 50]. Most of the prior works employ some form of *way-partitioning* where each way or column of a w-way set-associative cache is assigned to one of the cores. This allocation could be structurally enforced by physically constraining all cache lines belonging to a core to reside in a fixed subset of columns, or the allocation can be logically enforced by ensuring that core_i occupies no more than π_i lines per set (although those π_i lines may physically reside in any of the actual columns of the cache).

One common approach for adaptive cache partitioning is to use hardware to monitor the benefit or *utility* of allocating different numbers of ways to each core, and then to choose a partitioning to optimize some performance metric, such as minimizing the global number of cache misses. Figure 10 shows the number of additional hits that could be achieved for different cache allocations for two example programs. Based on this information, an allocation of three ways for core_0 and five ways for core_1 would result in the maximum number of misses avoided. While several past techniques are similar in spirit, in this work we compare against the recently proposed utility-based cache partitioning (UCP) approach [45]. UCP uses a set of *shadow tags* to track what the contents of the cache would be if each core had sole ownership of the entire cache. A hit in the i^{th} most recent position in the LRU stack indicates that an i-way set-associative cache would have provided a hit, but a lower-associativity cache would not have. By counting the number of hits corresponding to

each recency position, UCP can easily approximate associativity-benefit curves like those shown in Figure 10. Using this information, UCP then chooses an allocation to minimize global misses.

A potential limitation of strict way-partitioning of caches is that some cache capacity may be unutilized. If, for example, one cache set is not ever accessed by core_i, then the π_i cache lines allocated in that set go to waste. Another core_j that does access that set with some regularity will still be trapped in its own partition of π_j lines, unable to ever make use of those wasted resources. Some previous approaches include some facilities for recouping these otherwise unused cache lines [18, 46].

4.1.2 Deadtime Management

In a way-partitioned cache, the inefficient use of the unused cache lines discussed above are a direct artifact of the partitioning mechanism. Caches (whether for single- or multi-core processors) are not always efficiently used due to *dead* lines. Some cache lines are inserted into the cache, reused only a few times (perhaps never reused at all), and then eventually get evicted. There is an opportunity cost that from the time of a line's last access until it is evicted, the line consumes cache capacity without providing any benefit. In an LRU-managed cache, this "dead time" may last for hundreds or thousands of cycles as several other lines may need to be evicted before this dead line becomes the least recently used line.

In a way-partitioned cache, the opportunity loss can be magnified because dead lines may belong to other cores, but the partitioning mechanism prevents one core from stealing lines from other cores. Consider an example where two cores share a cache, and core_0 has one dead line. Under conventional way-partitioned cache management, when core_1 must make a replacement, it simply chooses a victim from among its own lines. In this example, however, if core_1 could choose core_0's dead line, then core_1 could potentially improve its hit rate while having no adverse impact on core_0, since core_0 would have eventually evicted its dead line without having derived any more hits from it anyway.

One particularly important and common case of dead lines are those lines that are dead

on arrival. The recently proposed *Dynamic Insertion Policy* (DIP) technique can insert lines directly into the least-recently-used position to minimize the residency time of such dead-on-arrival lines [44]. The thread-aware dynamic insertion policy (TADIP) uses dynamic monitoring of the policies combined with awareness of how these policy decisions interact in a multi-core context to reduce the amount of time dead lines take up the valuable shared cache resources [27].

4.2 Insertion And Promotion For Controlling Cache Occupancy

In this section, we first review cache insertion policies, introduce the concept of promotion policies, and then detail how we combine these to manage a shared cache.

4.2.1 Cache Insertion and Promotion Policies

Traditional cache management has focused on cache *replacement* policies. When a new line must be installed, the replacement policy chooses a victim or evictee. Most conventional systems make use of a *Least-Recently Used* (LRU) replacement policy or an approximation thereof [17]. Figure 11(a) illustrates an example cache set with eight lines, logically organized left-to-right from highest priority (8: keep in the cache) to lowest priority (1: to be evicted).[1] For LRU replacement, the priority ordering is equal to the access recency (the least recently used line has the lowest priority for retention). An access to line I causes line H (in the lowest priority) to be chosen as the evictee. In a conventional LRU-managed cache, the newly installed line is inserted in the highest priority (MRU) position. It has been observed that there are cache lines that are accessed only once and then never accessed again [29, 44]. By installing these lines in the highest priority positions, LRU actually *maximizes* the amount of time that these lines occupy the cache. Qureshi et al. introduced the concept of separating a cache replacement policy into independent victim selection and insertion policies [44]. As shown in Figure 11(b), for no-reuse lines, insertion of this particular line into the lowest priority position is actually much better as this minimizes the amount

[1] We use the terms lowest priority and highest priority instead of LRU and MRU, respectively, because once the insertion and promotion policies are modified, the ordering of the lines no longer strictly follows true "recency" and so calling a line "least recently used" when it is in fact *not* the least recently used line is inaccurate and can be confusing. Although the lines are shown left-to-right in priority order for illustration, the physical order in a cache may differ.

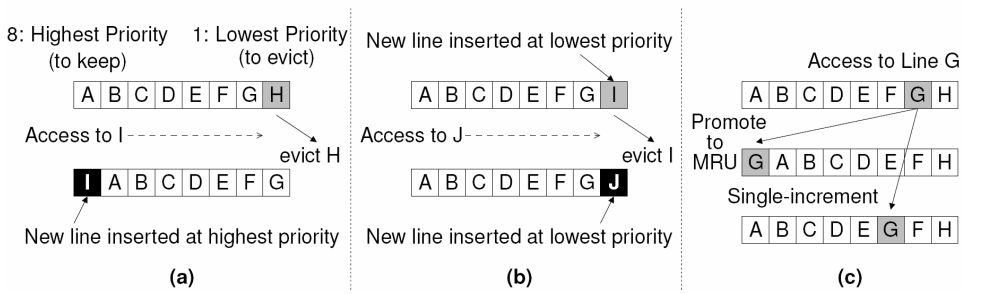

Figure 11: Example of (a) conventional insertion at the highest priority ("MRU") position and (b) insertion at the lowest priority ("LRU") position, and (c) different promotion policies.

of time that the line spends in the cache.

Independent of the victim selection and insertion policies, the conventional LRU-based policies all behave the same on a cache hit. That is, on any cache hit, the line is automatically moved or *promoted* to the highest priority position. We decompose cache management policies into three components: the victim selection and insertion decisions, as described earlier, and now the *promotion policy*. The promotion policy decides for a line that provides a cache hit how to update that line's position in the replacement priority order. Figure 11(c) illustrates both a traditional "promote-to-MRU" policy and a simple "single-increment" promotion policy.

4.2.2 Basic PIPP

We now explain our algorithm for Promotion/Insertion Pseudo-Partitioning (PIPP) of shared caches. For n cores, we will assume that we are given a target partitioning $\Pi=\{\pi_1, \pi_2, ..., \pi_n\}$ such that $\sum_{i=1}^{n} \pi_i=w$, where w is the total set associativity of the cache. In this work, we make use of UCP's utility monitors to compute the target partitions, but they could potentially be specified by the operating system [46] or from many other approaches. The three policy decisions for insertion, promotion and eviction are described in turn. On insertion, core$_i$ simply installs all new incoming lines at priority position π_i. That is, a core's partitioning allocation determines its insertion position. On a cache hit, the promotion policy for PIPP promotes the line by a *single* priority position with a probability of p_{prom}, and the priority is *unchanged* with a probability of 1-p_{prom}. Finally, the victim selection always

chooses the line in the lowest-priority position; this logic remains unchanged compared to conventional LRU.

PIPP does not strictly enforce the target partitioning, but the *combination* of targeted insertion and incremental promotion creates results *similar* to explicit partition enforcement, hence *pseudo*-partitioning. For systems with more than two cores, each core_i still installs its lines into the priority position determined by π_i. While this insertion policy tends to cluster cache lines near the low-priority end of the ordering (for example, a quad-core system with a 16-way cache and Π=$\{6, 4, 4, 2\}$ results in no lines ever being inserted with a priority higher than 6), this approach can still produce the desired target partitioning. Core_0's lines experience less promotion/demotion competition than the other cores, and so its lines tend to stay in the cache and are more likely to get promoted into higher priority positions. Core_1 and core_2 get inserted at the same position; while they will directly compete for cache resources, neither has a distinct advantage over the other, and both have a disadvantage compared to core_0 by being inserted at a lower priority position, so statistically both end up occupying fewer lines than core_0 but about the same lines as each other. Finally, core_3 must "swim upstream" against *all* of the traffic from the three other cores, and ends up occupying the fewest lines. Again, PIPP does not explicitly enforce the partitioning, but this example illustrates how partitioning-like behavior can be induced.

4.2.3 Example

Figure 12 illustrates a simple example for an eight-way cache shared between two cores with Π=$\{5, 3\}$. Core_0's cache lines are represented by numerals in squares, and core_1 by letters in black circles. The figure also includes the insertion positions for each core as determined by the partitioning. Core_1 makes a request for line D, which misses and is inserted at position 3. Similarly, core_0's request for lines 6 and 7 both miss and are in turn inserted at position 5. The next access is for core_1's D, which hits in the cache. In a normal promote-to-MRU scheme, line D would be promoted to the highest priority position, but in our PIPP scheme, we promote the line by only a single position (for this example, we simply assume that p_{prom}=1). The example continues with several more misses (insertions)

and hits (promotions).

The example contains a few interesting sections. Note that for most of the time, the actual cache occupancy for each of the two cores matches that of the target allocation $\Pi=\{5,3\}$. There are intervals, however, where the instantaneous occupancy does deviate from the target partitioning, thus highlighting the fact that our scheme does not explicitly *enforce* partitions. To understand how the insertions and promotions can control capacity, consider a core with a marginal utility curve where most of the core's hits can be achieved with three ways, and that adding a fourth way does not provide many additional hits. If this core manages to grab a fourth way under PIPP, this cache line will not provide many additional hits (by assumption of the utility curve), the other more useful lines will continue to provide hits and get promoted above the extra line and therefore push the low-utility fourth line down the priority ordering where it will be quickly evicted, at which point the cache occupancy reconverges to that of the target allocation.

In addition to capacity management, the example in Figure 12 also illustrates some similarities and differences between PIPP and TADIP. Consider line E and assume that it does not exhibit any reuses. Inserting E in priority position 3 reduces the amount of time this dead line occupies cache space compared to conventional MRU insertion. Note that for line E, PIPP's approach is not as effective at eliminating dead time as TADIP, which would have inserted E directly into the lowest-priority position, thereby minimizing the amount of time the dead line spends in the cache. Consider line D which exhibits one reuse before becoming dead. On reuse, TADIP would promote the line directly to the highest priority position, but at this point the line is now dead and so this approach actually maximizes the residency time of the dead line. On the other hand, PIPP only promotes line D by a single position, thereby keeping D in a lower priority position which allows it to be evicted that much more quickly.

4.2.4 Stream-Sensitive PIPP

Poor cache performance due to inter-core interference is often caused by one or more programs that exhibit memory access patterns with very poor locality. Many of these situations

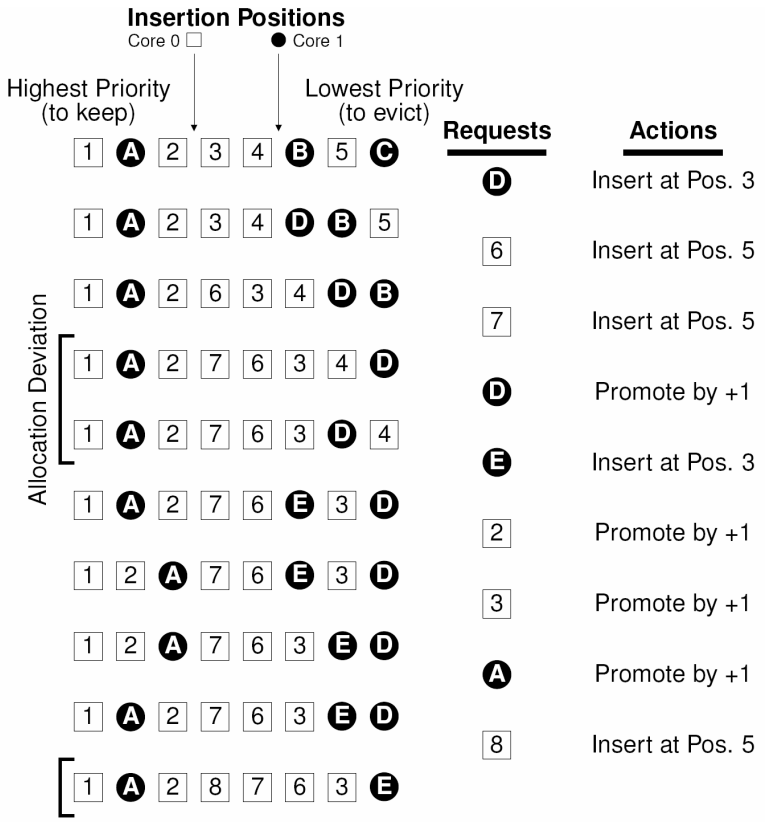

Figure 12: Example operation of PIPP for a variety of cache misses (insertions) and hits (promotions). Evictions always choose the lowest-priority cache line.

Table 2: Dual-core and quad-core workloads used in our evaluation. In the UCPx workloads, UCP performs better than TADIP, and visa-versa for the DIPx workloads.

Dual-Core Workloads		**Quad-Core Workloads**	
Workload Name	Application Suites, Names and Inputs	Workload Name	Application Suites, Names and Inputs
UCP2-0	f00:art, FP:ebgm.l	UCP4-0	i06:hmmer.r, f00:art, f00:equake, f06:soplex.r
UCP2-1	PB:continuous, f00:equake	UCP4-1	f00:art, Mi:dijkstra, FP:ebgm.l, f06:lbm
UCP2-2	i00:eon.k, md:mpeg4.d	UCP4-2	i00:crafty, i06:hmmer.r, f00:art, i06:omnetpp
UCP2-3	i06:h264ref.f, i06:gcc.g	UCP4-3	i06:hmmer.r, i00:bzip2.g, i06:astar.r, f06:bwaves
UCP2-4	i06:perl.s, i06:hmmer.n	UCP4-4	FP:ebgm.e, i06:h264ref.f, md:jpg2000.d, f00:art
UCP2-5	md:mpeg2.d, f06:sphinx3	UCP4-5	i06:hmmer.n, i06:bzip2.p, f06:soplex.p, f06:bwaves
UCP2-6	i06:perl.s, f06:sphinx3	UCP4-6	i06:h264ref.f, i06:gcc.g, md:pegwit.e, i06:hmmer.r
DIP2-0	i00:eon.k, FP:ebgm.e	DIP4-0	FP:ebgm.l, md:jpg2000.d, i00:eon.c, md:jpeg.d
DIP2-1	FP.ebgm.e, BP:clustalw.c	DIP4-1	BP:clustalw.c, Mi:adpcm.d, MN:bayes, FP:ebgm.e
DIP2-2	md:h264.e, FP:ebgm.l	DIP4-2	i06:h264ref.s, i00:gcc.s, f00:equake, i00:mcf
DIP2-3	FP:ebgm.l, md:jpg2000.d	DIP4-3	md:pegwit.d, FP:ebgm.e, i00:eon.k, md:pegwit.k
DIP2-4	i00:eon.k, i00:mcf	DIP4-4	i00:eon.k, i00:mcf, md:pegwit.d, md:adpcm.d
DIP2-5	i06:mcf, md:jpg2000.d	DIP4-5	i06:sjeng, i00:eon.r, FP:ebgm.e, i00:eon.k
DIP2-6	f00:art, md:jpg2000.d	DIP4-6	i06:mcf, md:h264.e, md:jpg2000.d, Mi:adpcm.d

The prefix before the colon identifies the benchmark suite; f00/f06/00/i06 are SPECcpu fp2000, fp2006, int2000 and int2006, respectively; BP is BioPerf [5]; FP is FacePerf [49]; md is MediaBench [33]; and Mi is MiBench [19]; MN is MineBench [41]; PB is PhysicsBench [60].

have stream-like behaviors characterized by both a high access frequency as well as a large number of cache misses. These "low-utility" applications insert a large number of lines in the cache, often at a very high rate relative to the access frequencies of the other cores, and as a result they quickly flush out the cache lines used by the other cores. The worst part of it is that the useful lines evicted are replaced by useless lines that do not get reused.

Fortunately, such cache-unfriendly behaviors are often easy to detect. We propose a simple modification to PIPP that accounts for applications that exhibit stream-like behaviors. For each core$_i$, we make use of the Utility Monitor shadow tags to track (1) A_i the total number of accesses by core$_i$, and (2) m_i the number of misses the core would experience *if it had access to the entire cache for itself.* If the total number of misses exceeds a certain threshold $m_i \geq \theta_m$ or if the miss rate $\frac{m_i}{A_i} \geq \theta_{mr}$, then PIPP assumes the core is running a stream-like application. The intuition is that a large number of absolute misses will likely cause significant thrashing and interference with other cores, and a high relative miss rate means that even if more lines could be obtained, there would be very little return on the investment.

When PIPP detects that a core is running a stream-like application, it modifies its behavior as follows. First, all insertions for the streaming core$_s$ are made at priority position π_{stream}, independent of the target partition π_s. We set this insertion position to equal the current number of stream-like applications, effectively "allocating" a single way to each such program. The idea is that since the streaming accesses are very unlikely to exhibit reuse, there is no point in inserting them with priority greater than π_{stream}. Next, promotion for hits due to core$_s$ only occur with a reduced probability of $p_{stream} \ll p_{prom}$. The greatly reduced promotion probability ensures that *only* those cache lines that can demonstrate significant reuse will have a reasonable probability of getting promoted to higher priorities where they have a better chance of staying in the cache. This has similar properties to statistical filtering of caches [6]. For the corner case where *all* programs simultaneously exhibit streaming behavior, PIPP reverts to inserting all lines at the highest-priority position since there are no non-streaming applications to hurt.

4.3 Experimental Evaluation

This section first briefly explains our simulation methodology, and then presents our main performance results.

4.3.1 Simulation Methodology

Our cycle-level model is built on top of the SimpleScalar toolset for x86 [4, 36]. We base our processor configuration on the Intel Core 2 processor clocked at 3.2GHz [13]. The pipeline has a minimum branch mispredict penalty of 14 cycles, the in-order portions (decode/commit) are four-wide, and the out-of-order core can issue up to six μops per cycle. We use a 96-entry ROB, 32-entry RS, 32-entry LDQ and a 20-entry STQ. Each core has 32KB, 8-way, 3-cycle level one instruction and data caches, and all cores share a 4MB, 16-way, 11-cycle LLC. All caches have 64-byte lines, and all are equipped with hardware prefetchers. We model a SDRAM memory with 9-9-9 timing on a 800MHz front-side bus (effective 1.6GHz with DDR2).

We use multi-programmed workloads created from a mix of applications including SPECcpu with the reference input sets as well as a wide variety of other suites. We used SimPoint 3.2 to choose representative samples [23]. The workloads are classified into two groups: those for which UCP performs better than TADIP, and those for which TADIP works better than UCP. Table 2 lists all of the dual- and quad-core workloads.

We also evaluated many additional workloads with less interesting behaviors (e.g., working sets mostly fit in the LLC, high DL1 hit rates resulting in low LLC activity) to verify that the techniques do not inadvertently cause any significant performance slowdowns; for brevity, these results are omitted since they do not show any unexpected results.

For each workload, we warm the caches and branch predictors for 500 million instructions, and then perform detailed simulation for 250 million instructions per benchmark. When an application reaches its instruction limit, it continues executing to compete for cache resources, but the statistics that we report only correspond to the original 250M instruction sample; this methodology is consistent with prior studies [27, 45]. Since one

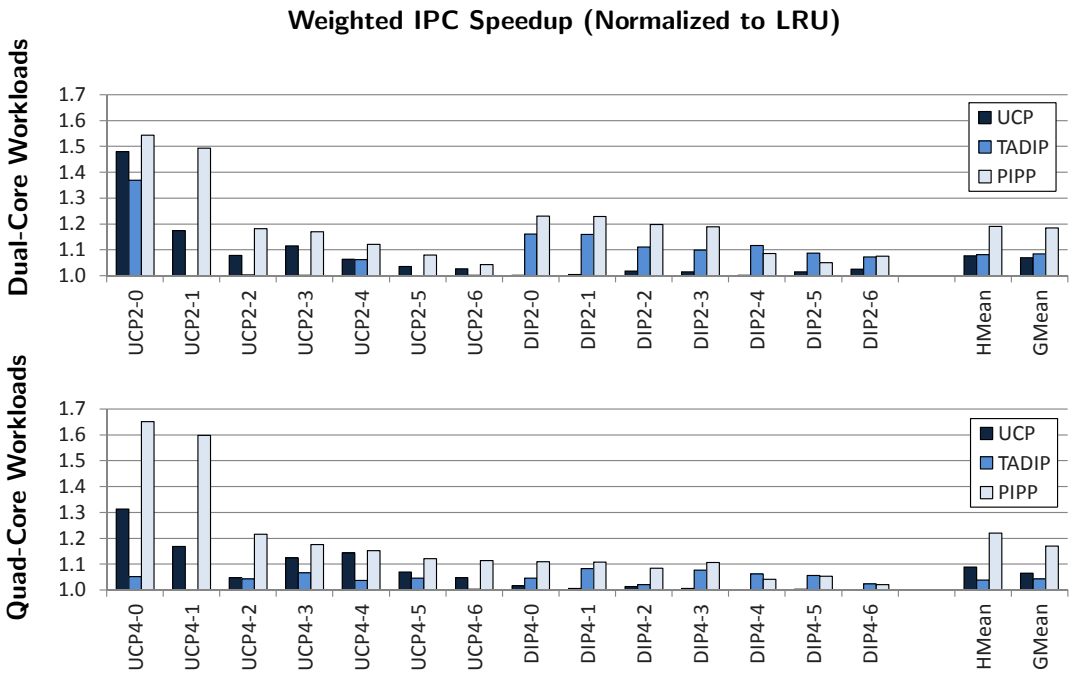

Figure 13: Performance as measured by the weighted speedups of IPC for UCP, TADIP and PIPP normalized to an LRU-managed cache for dual-core and quad-core workloads.

benchmark may take a lot longer to reach its instruction limit than others, the total number of simulated cycles is typically much larger than what might be expected for 250 million instructions per core. On average, each experiment simulated over one billion cycles (not including warm-up activities) with a maximum of about 9.5 billion cycles for one of the quad-core workloads with substantial memory accesses. For all experiments, we report weighted speedup (i.e., SMT speedup) $\sum_{i=1}^{c} IPC[i]/IPC_{sa}[i]$, where IPC_{sa} is the *s*tand-*a*lone IPC when the core has exclusive access to the entire processor [37], IPC throughput $\sum_{i=1}^{c} IPC[i]$, and the harmonic mean of weighted speedups $\frac{c}{\sum_{i=1}^{c}(IPC_{sa}[i]/IPC[i])}$ which accounts for both fairness and performance [9]. Unless otherwise specified, "performance" will refer to weighted speedup.

4.3.2 Results

For all performance comparisons, we use a conventional unmanaged, shared cache with LRU replacement as the baseline. We compare against UCP, TADIP (specifically TADIP-F) and PIPP. Both UCP and PIPP make use of shadow tags with dynamic set sampling to track per-core utility curves (32 sets per shadow tag) [45]; both UCP and PIPP use

this information to feed to the same partitioning algorithm (optimal for dual-core, and Lookahead [45] for quad-core) to select the target partition [45]. All PIPP results here make use of the stream-sensitive version of PIPP with probabilities p_{prom}=$\frac{3}{4}$ and p_{stream}=$\frac{1}{128}$. The probability p_{prom} simply requires generating a two-bit pseudo-random number and testing that the result is not equal to zero, and p_{stream} requires generating a seven-bit pseudo-random number and testing that it *is* equal to zero. The stream-detection thresholds were likewise chosen for easy implementation. We use $\theta_m \geq 4095$, which simply requires testing that a 12-bit saturating counter has reached its maximum value. Similarly, θ_{mr} can be selected for easy computation; we used the value 12.5%: $\frac{m_i}{A_i} \geq \frac{1}{8}$ is equal to $m_i \geq \frac{A_i}{8}$ (right-shift A_i by three and compare with m_i).

Figure 13 shows the performance impact of the different cache management techniques for the weighted IPC speedup. For the dual-core simulations, PIPP consistently outperforms unmanaged LRU by a large margin (19.0% on the harmonic mean), and also outperforms both UCP and TADIP (10.6% and 10.1%, respectively). Similar results hold for the quad-core case where PIPP is 21.9% better than LRU, 12.1% better than UCP and 17.5% better than TADIP.

Figure 14 shows the results measured by total IPC throughput and fair speedup, relative to LRU. The trends are very similar to the weighted speedup results, demonstrating that PIPP also provides higher raw throughput and better fairness. Due to the similarity of the overall trends, we only deal with weighted speedup in the rest of this chapter.

PIPP consistently outperforms UCP for both dual-core and quad-core workloads on all of the performance metrics (with the one exception of UCP4-4 for the fair speedup metric where the performance of PIPP is still very close to UCP). PIPP's strong performance comes from its effective capacity management combined with it not being strictly bound to the partition allocations and its abilities to exploit DIP-friendly behaviors. These attributes will be further explored in the next section.

There are a few workloads where, while still performing well compared to LRU, PIPP still gets beat by TADIP. An interesting pair of workloads to contrast are DIP2-3 and DIP2-5, where PIPP performs better than TADIP on DIP2-3 and TADIP is superior on DIP2-5, and

each responds well to LRU insertion. For each workload, one of the benchmarks contained a large number of lines that observe a single reuse in an unshared cache; contention in a shared cache shortens the lifetimes of these lines such that they become zero-reuse lines that TADIP takes advantage of.

In the case of DIP2-3, there are many lines with just a few uses, but very few lines with many uses. It would be desirable to have a cache management policy that can keep the lines resident in the cache just long enough to expose these additional hits, but then quickly evict them after they become dead. PIPP can provide this type of effect because it inserts the lines with slightly higher priority than the lowest possible which provides a short window for additional hits to manifest. PIPP's incremental promotion policy discourages these lines from staying in the cache for too long after they become dead. TADIP on the other hand may not keep the lines in the cache long enough to expose the extra hits, and when it does, the lines are directly promoted to the highest priority position. Since most lines in this program have only a single reuse, TADIP's promotion policy actually ends up maximizing the dead time of these lines.

Compared to the DIP2-3 workload, DIP2-5 (where TADIP performs better) has many more lines with many more reuses (in addition to the many zero-reuse lines). In this scenario, PIPP's incremental promotion policy is too cautious leading to situations where lines with more uses in the near future do not get promoted high enough in the priority ordering for them to evade eviction before their next uses. TADIP's more aggressive promotion policy does a better job at keeping these lines in the cache to expose many more hits. This suggests that perhaps PIPP could be further enhanced by providing some facilities to dynamically tune the aggressiveness of the promotion policies.

4.3.3 Analysis of PIPP

We have claimed that PIPP is effective at controlling the occupancy of shared caches, but so far we have only demonstrated that the performance of PIPP is as good as or better than UCP. To measure the effectiveness of PIPP's capacity management, we measured the difference between each core's actual cache occupancy and the target partitioning. We

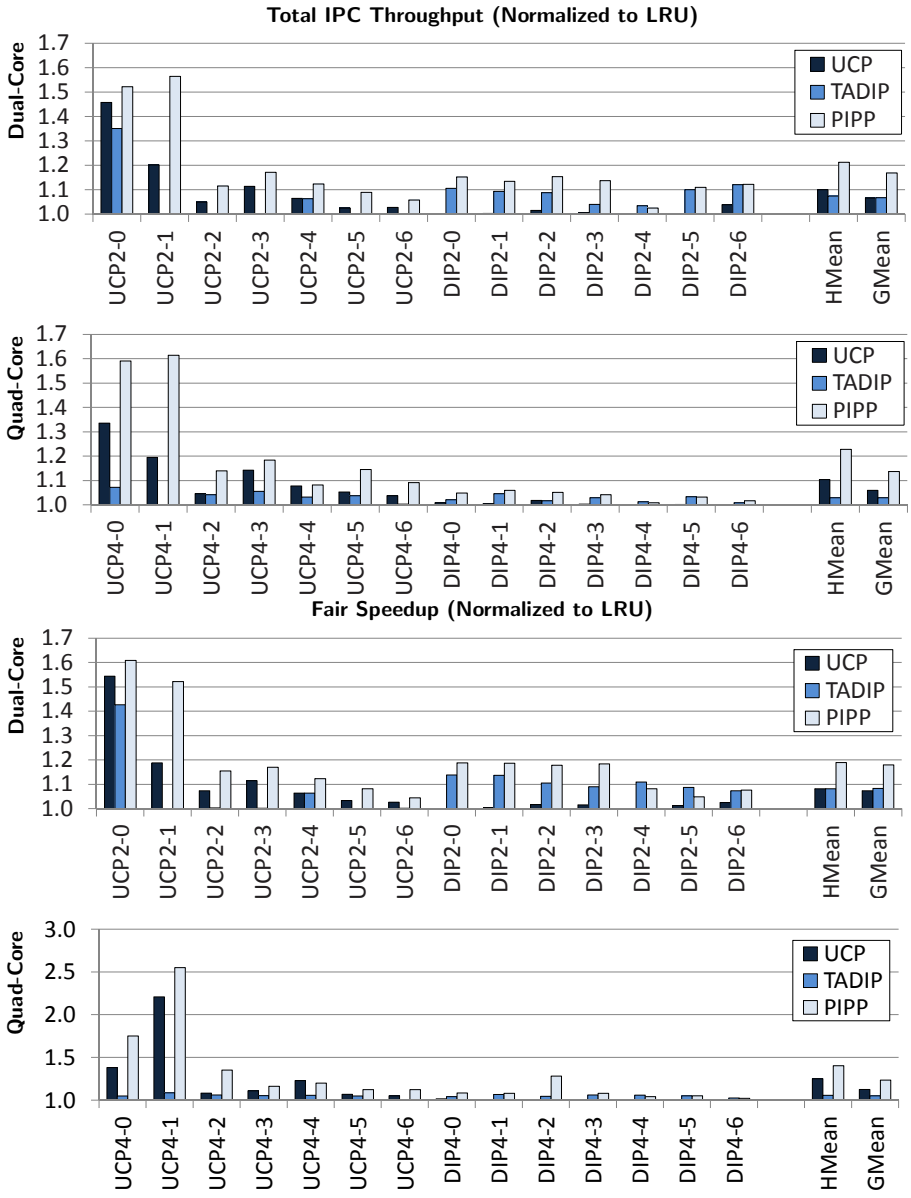

Figure 14: Performance results for the IPC throughput and fair speedup metrics.

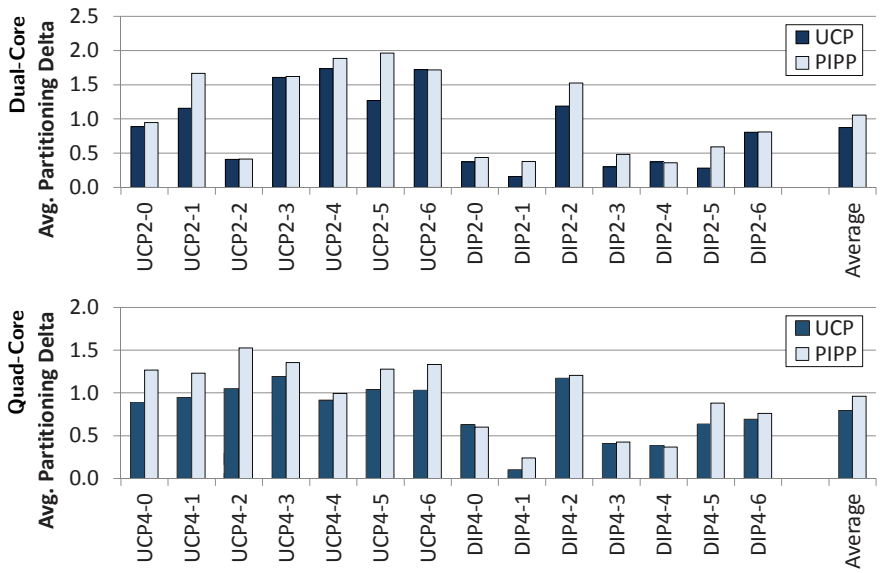

Figure 15: Average partitioning deviation for all of the dual-core and quad-core workloads for PIPP.

define a core's partitioning deviation as the absolute difference between the average number of ways occupied and the target number of ways allocated. For example, if core$_i$ has a target allocation of π_i=3 in an eight-way cache, but actually occupies half of the cache's total capacity, then on average core$_i$ occupies four out of every eight lines in the cache, and so the partitioning deviation is equal to one. During simulation, every one million cycles we measure the partitioning deviation of each core and then we average all of the samples over the entire simulation.

Figure 15 shows the average partitioning deviation for all of the workloads. For the vast majority of the workloads, the partitioning deviation is within 1.0 of the target allocation. This shows that PIPP can create aggregate conditions that are similar to those created by UCP. The imperfect partitioning is a direct result of the fact that PIPP only *pseudo*-partitions the cache but does not explicitly enforce allocations. Nevertheless, PIPP still manages to balance per-core capacities in a way that comes reasonably close to the target allocations. It is also important to note that PIPP's partitioning deviation does not necessarily result in lower performance, for example due to the theft of a dead line from another core.

PIPP can reduce the amount of time that dead-on-arrival lines reside in the cache by

simply inserting them in a position of lower priority. Of particular interest are the workloads DIP2-0 and DIP2-1 where UCP provides absolutely no benefit and TADIP is still able to deliver about 16% speedup for both workloads. These "pure TADIP" workloads exhibit a large number of lines with no-reuse, and this is reflected by the fact that on average, TADIP inserts lines with a priority of 1.683 and 1.685 (where a priority of 1 is "LRU" insertion, and a priority of 16 is "MRU" insertion) for DIP2-0 and DIP2-1, respectively. For the same workloads, PIPP has average insertion priorities of 1.330 and 1.329, effectively showing that PIPP can mimic TADIP's LRU-insertion behavior.

One of the qualitative differences between PIPP and conventional way-partitioned cache management schemes is that the partitioning is not rigid which allows cores to obtain more cache resources than would be normally allowed. In particular, we say that a core "steals" a cache line when it inserts a line into the LLC, but inserting this line causes the core to exceed its target partition (in this cache set). For example, if a core has a target allocation of π=3 ways, inserts a line and then the core now occupies four or more ways in this set, then we say that this insertion stole a cache line.

For each workload, we monitor every LLC insertion and record whether the insertion resulted in a line theft. We then monitor two types of events. The first is for every line which was stolen, how many times was that line subsequently reused? This provides an estimate of the benefit of stealing capacity from other cores. Second, for every stolen line, we remember the previous occupant's address. If we later observe a miss on this address, but we find the address in one of the stolen line's previous-address fields, then we record this as a miss that was caused by stealing the line ("forced miss"). Figure 16 shows the averages (across accesses by all cores) for these metrics along with markers indicating the net impact accounting for both additional hits and misses caused by line thefts. Note that these metrics are not necessarily directly proportional to performance impact because, for example, the number of hits credited for a stolen line include all reuses of that line. In a conventional cache, it is possible that after the first access, the line is reinstalled in the cache and the remaining hits would have occurred anyway. Nevertheless, the results show that the lines that are stolen are indeed useful, and that the theft of lines from other cores

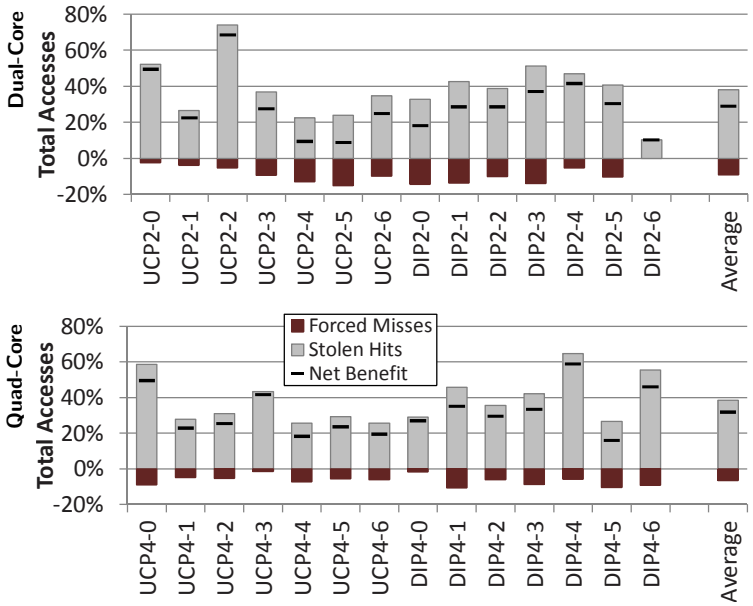

Figure 16: Number of hits on lines stolen from other cores, number of misses induced by having lines stolen by other cores, and the difference.

do not introduce that many more misses. This result is intuitive in that when a line gets stolen, the evicted line is already in the lowest priority position in the eviction order, which tends to be correlated with a lack of near-future reuses. That is, when one core steals a line from another core, there is a good chance that the line would have been evicted soon anyway. PIPP can effectively make use of these dead lines to improve the efficiency of the cache.

4.3.4 PIPP Parametric Sensitivity Analysis

Our PIPP mechanism contains several parameters that can be chosen by the computer architect. These include decisions about promotion policies, whether to make PIPP aware of streaming application behaviors, and setting the two different probabilities p_{prom} and p_{stream}. We first consider several variants on the PIPP mechanism. These are summarized in Table 3, with the original PIPP included for reference. These variants were chosen to demonstrate the importance of different design choices for PIPP. Figure 17(a) shows the performance degradation (higher is worse) for the harmonic mean of the weighted IPC speedup normalized to the stream-sensitive PIPP. Omission of any of these features can

Table 3: Variants on PIPP for studying the importance of different components of the algorithm.

Configuration	p_{prom}	p_{stream}	Promotion
Baseline PIPP	3/4	1/64	0/+1
50-50 Probability	1/2	1/2	0/+1
Always MRU Promotion	1	1	MRU
Stochastic MRU Promotion	3/4	1/64	0/MRU
Always Promote by +1	1	1	+1
No-Stream	3/4	3/4	0/+1

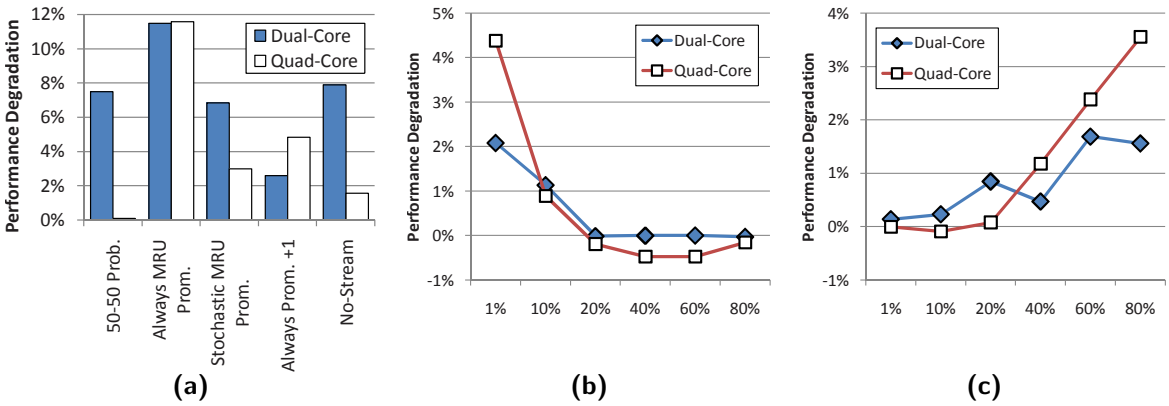

Figure 17: Performance sensitivity to different (a) PIPP design choices, (b) values for $\mathbf{p_{prom}}$ and (c) values for $\mathbf{p_{stream}}$.

lead to an 11.6% performance penalty.

We also explored the sensitivity of the results to different choices of p_{prom} and p_{stream}. It would not be desirable to have to carefully retune these probabilities for different workloads or new processor designs. Figure 17(b) shows the performance results when all parameters for the original stream-sensitive PIPP are held constant except for p_{prom}. The results are normalized to the original case where $p_{prom} = \frac{3}{4}$. Note that for any similar values for p_{prom}, the overall performance change is less than 1%. Similarly, Figure 17(c) modifies p_{stream} while keeping all other parameters the same. The original value for p_{stream} was $\frac{1}{128}$, but these result show that one can change this probability over a reasonably wide range and the overall performance changes are quite small. These results are good in that they suggest that the probability parameters need not be chosen too carefully; any reasonable values will result in good performance.

4.4 Implementation Issues

In this section, we briefly discuss some of the remaining hardware overhead required to implement PIPP, and then present a simple optimization to eliminate much of the remaining costs.

4.4.1 Hardware Overhead

One of the critical components of many previously proposed cache partitioning approaches is the mechanism for estimating or predicting what the benefit/cost would be of providing a core with more/fewer ways. This information provides the input for the partitioning mechanism to make its decisions. In particular, the implementation of PIPP evaluated thus far in this chapter simply makes use of the same Shadow Tag approach used in the UCP work and discussed in the motivation section. As described earlier, the *Utility Monitor* (UMON) maintains one set of shadow tags per core to track what the cache's contents would be if each core had exclusive access to the cache. Depending on the recency positions of the hits observed in the shadow tags, utility monitoring counters are updated to create the corresponding utility curves such as those shown in Figure 10.

The UMON shadow tags represent additional overhead that would not be required in a conventional unmanaged cache, although the use of Dynamic Set Sampling (DSS) reduces the overhead by a significant amount. For example, a cache with 4096 sets shared by four cores requires 1.1MB to store the unsampled shadow tags (assuming 36-bit tag entries), whereas with DSS the overhead is reduced to only 9.1KB (assuming 32 sampled sets). Suh et al. proposed to estimate marginal utility based on the actual hit position in the shared cache [51]. A hit in recency position i in the real cache causes the i^{th} marginal utility counter to get incremented. For example, a set containing the lines {A, B, 1, C} where A is the most recently used line and C is the least, and A, B and C belong to core_0, a hit on line C causes core_0's fourth counter to be incremented because C is located in the fourth most-recently used position, but if core_0 had the entire cache to itself, C would only be in the third most-recently used spot. It is clear that while this approach does not incur any additional storage overhead for shadow tags, the contents of the marginal gain counters

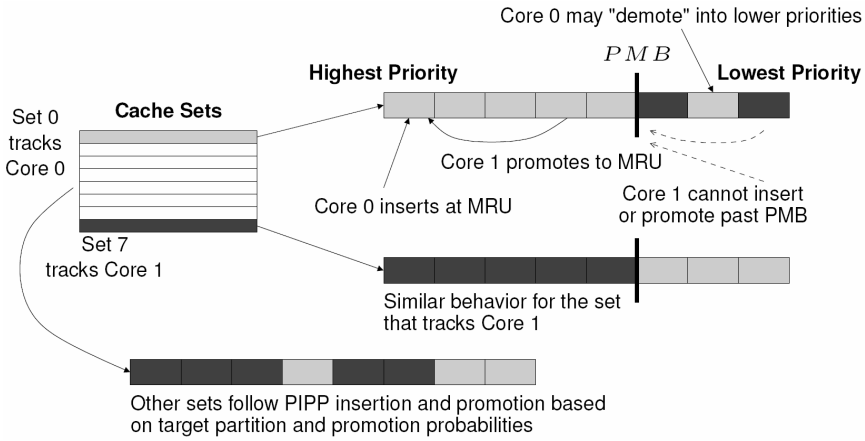

Figure 18: Example organization of the In-Cache Estimation Monitors (ICEmon) for two cores sharing an eight-set, eight-way cache.

may be compromised, leading to inaccurate utility curves [45].

4.4.2 Elimination of Shadow Tags

We propose In-Cache Estimation Monitors (ICEmon) which can be thought of as a hybrid of Suh et al.'s approach, Qureshi and Patt's UMON, and Qureshi et al.'s *leader set* idea [44]. A small number of sets use a modified cache management policy to facilitate utility tracking. Figure 18 shows an example cache with eight sets, where one set is reserved for tracking core_0 and another for core_1. Set zero estimates the utility for core_0. For all accesses by core_0, this set is managed in a conventional LRU manner (evict the LRU line, insert at the MRU position, promote to the MRU position on a hit). We then enforce a *private monitoring boundary* (PMB) where lines inserted/accessed by other cores are not permitted to obtain a recency position greater than PMB. Core_1 continues to follow its insertion and promotion policies, with the modification that insertion and promotion are capped at PMB, as shown in Figure 18. Set seven monitors core_1 in a similar fashion with core_0 not being able to cross the PMB.

By dedicating w–PMB ways (in a w-way cache) to the monitored core, our ICEmon is able to accurately track the marginal gain updates for the first w–PMB ways. These ways are typically the most important in that they account for the majority of the area underneath the marginal utility curves (i.e., the first few ways usually provide the most

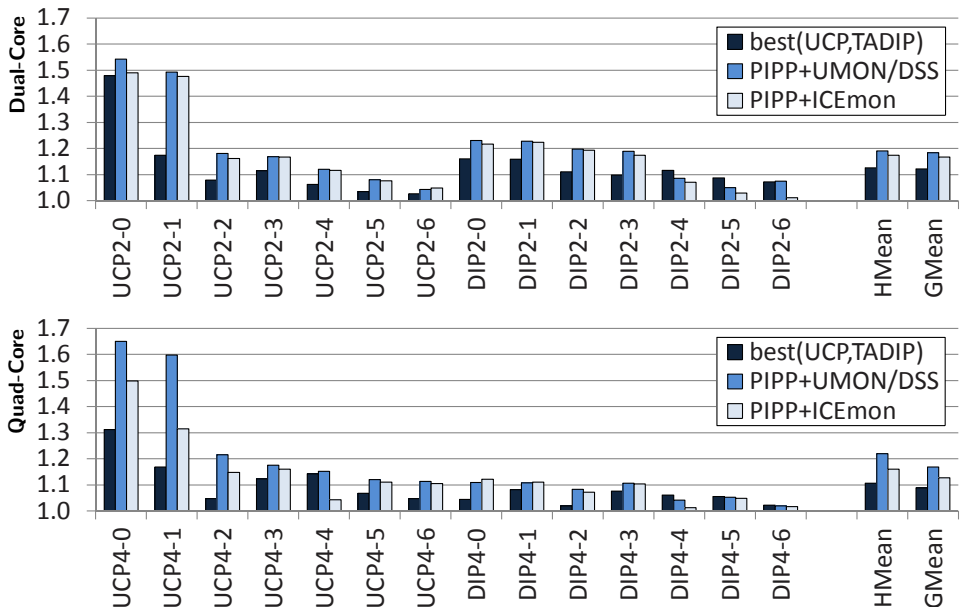

Figure 19: Weighted speedup of PIPP with different utility tracking mechanisms.

hits). For the remaining w ways, the utility tracking may have some errors because the lines accessed by the other cores will pollute and perturb the recency ordering of the core being tracked. This can create problems similar to Suh et al.'s monitoring scheme, but the impact on ICEmon is much less because these tracking errors only impact the PMB-least recently used lines which typically account for a much smaller number of the total accesses.

Figure 19 shows the performance of PIPP using UMON with DSS, and PIPP using our proposed ICEmon scheme. For reference, we also include the performance of the *best* of UCP and TADIP. The PIPP+ICEmon configuration uses 32 leader sets per core to estimate the marginal gain counters. For each leader set, PMB is set to four ways.

Our ICEmon mechanism for estimating the marginal utilities does introduce some error in the partitioning decisions, as shown by the difference in the PIPP+UMON/DSS and PIPP+ICEmon results. For the dual-core configurations, the small amount of error introduced by ICEmon, as compared to UMON/DSS, results in an average performance loss of only 1.4% (for both means); for the quad-core results, the performance loss is a more modest 3.6%/4.8% for the harmonic/geometric mean. Note that even with the reduction in performance due to the less accurate ICEmon scheme, the average performance across all

Table 4: Summary of overheads for different marginal utility estimation schemes. Example storage overhead assumes **s**=4096 sets, **w**=16 ways, **N**=4 cores, **t**=36 bits per shadow tag entry, $\alpha=\frac{\mathbf{1}}{\mathbf{128}}$ (DSS sampling rate), UMON counter size=10 bits.

	Shadow Tag Storage	UMON Counters	Storage 4MB/16-way L2, 4 cores
UMON (*no DSS*)	$swtN$	wN	1.1 MB
UMON (*w/ DSS*)	$\alpha swtN$	wN	9.1 KB
ICEmon	0	wN	20 bytes

of the dual-core and quad-core workloads still exceeds the best of both UCP and TADIP.

The main reason that the quad-core results are not better is that we did not recalibrate the PMB setting, and so in the leader sets, *three* cores worth of cache lines are forced to share the four ways which results in more error in tracking the marginal utility of the last few ways. We were also concerned that a hot set that mapped to a leader set could cause significant pathological behaviors. We considered a variant of PIPP/ICEmon where during each sampling interval, the leader sets are rotated so *all* sets in the cache take turns as leaders. It turns out that this hot-set phenomenon is not important, at least for our workloads, and this rotating leader-set approach made very little difference on performance. Table 4 summarizes the storage overhead required for the UMON and ICEmon approaches for estimating marginal utility.

Many previous studies have proposed a variety of mechanisms to improve the performance of shared last-level caches by exploiting a variety of common memory access behaviors. In this work, we have introduced a single unified technique that can provide the benefits of capacity management, adaptive insertion and inter-core capacity stealing. By covering multiple types of memory behaviors, our proposed PIPP scheme delivers higher performance than previously proposed techniques.

CHAPTER V

MANAGING THE SHARED CACHE BY CONTAINING THRAHSERS

In this chapter, we demonstrate that a simpler cache partition scheme can be derived based on our animal classification scheme from Chapter 2. We see that the main situations where dynamic cache partitioning is useful are when a devil-like application is present. Utility-based Cache Partitioning effectively does a good job at containing the devil application because UCP explicitly knows that allocating more ways to the devil provides very little benefit as measured by the marginal gains/utility. We observe that it is sufficient to detect that an application is acting like a devil, and then to contain it. As a result, this simple and effective approach is able to achieve similar performance benefit to UCP with much less overhead. This would directly support the thesis that the shared cache in multicore processors can be managed to improve overall system performance by adding low overhead hardware. Because the devil detection proposed in this chapter is very cheap to implement, it will especially support the "low overhead hardware" part of the thesis.

5.1 Thrasher and Non-thrasher

In this chapter, we group the turtle, sheep and rabbits into "Non-Thrasher" and the devil itself into "Thrasher" group. Since we no longer care about the exact criteria for recognizing rabbit behavior, we can also reduce the hardware requirements for our scheme. In particular, while we still need to maintain a set of shadow tags (with the same caveats as before regarding the use of set sampling to keep the overhead acceptable) to determine $Misses_{solo}$, we no longer need the per-core UMON marginal utility counters, as only the Rabbit classification makes use of the $WaysNeeded$ metric. The condensed algorithm for detecting devil is in Algorithm 4.

We use a variety of benchmarks from SPEC2000 and SPEC2006 from both the integer and floating point suites, PhysicsBench [60], MediaBench [15, 33], MineBench [41],

Algorithm 4 Simplified classification rules for thrasher and non-thrasher

if $((Accesses > \theta_{acc})$ AND $((MissRate_{solo} > \theta_{MR})$ OR $(Misses_{solo} > \theta_{miss})))$ **then**
  Classification = **Thrasher**
else
  Classification = **Non-Thrasher**
end if

Table 5: Baseline 4-wide processor configuration. All caches use 64-byte lines.

Parameter	Value	Parameter	Value
ROB Size	96 entries	RS Size	32 entries
LDQ/STQ Size	32/20 entries	IL1/DL1	32KB/8-way/3-cyc
Shared L2 (dual-core)	4MB/16-way/9-cyc	Shared L2 (quad-core)	8MB/32-way/9-cyc
Function Units	3 IALU, 1 IMul, 1 FAdd, 1 Div, 1 FMul, 1 Load, 1 STA, 1 STD		
Main Memory	SDRAM, 800MHz bus (DDR), 6-6-6, 3.2GHz CPU speed		

MiBench [19] and BioPerf [5]. For SPEC, We use reference inputs. Table 6 lists the applications and their baseline statistics. Applications with very low DL1 miss rates were not considered for workload creation because they have practically no impact on sharing/contention in the LLC (i.e., they are neither hurt by nor benefit from any of these techniques).

We run each benchmark and observe the fraction of time each is classified as exhibiting thrashing behavior. These results are tabulated in Table 6 along with some other basic information such as the baseline IPC, cache access frequency, and the performance difference between providing a 4MB cache versus only a 1MB cache. The list is sorted from the most-frequently thrashing to the least. Note that due to the simplicity of our classification scheme, we do not distinguish between applications that are moderately thrashing (e.g., ϵ more misses than θ_{miss}) and extremely thrashy (e.g., much more misses than θ_{miss}). Likewise, this classification does not differentiate between thrashing behavior caused by large working sets versus those exhibiting streaming behaviors.

We then created several workloads with different combinations of thrashing (T) and non-thrashing (N) applications, listed in Table 7. For the sake of workload creation, we consider any benchmark that spends >50% of the time exhibiting thrashing behavior as thrashing. We also include two more 1T3N workloads (F, G) that incorporate a few applications with small working sets to ensure that the proposed technique does not inadvertently hurt performance in such a situation. These additional "small" applications are taken from the

Table 6: Benchmark classification. APKI stands for accesses per thousand instructions. Codes: F0 (SpecFP'00), F6 (SpecFP'06), I0 (SpecInt'00), I6 (SpecInt'06), MI (MiBench), MD (MediaBench), MN (MineBench), PB (PhysicsBench), BI (BioPerf). Benchmarks N6-N18 spend <0.5% of the time thrashing.

Benchmark Name		Base IPC	4M/1M Slowdown	APKI	% Time Thrashing
T0	F6-milc	0.28	0.4%	60.9	100.0%
T1	F6-lbm	0.23	0.0%	14.1	100.0%
T2	F6-soplex	0.26	5.5%	87.5	99.4%
T3	F0-equake	0.32	45.8%	129.2	98.6%
T4	F6-sphinx3	0.40	3.1%	69.8	96.2%
T5	I6-gcc	0.60	0.6%	30.0	92.8%
T6	I6-libquantum	0.28	0.0%	149.5	63.2%
N0	MN-semphy	1.06	5.4%	1.3	37.5%
N1	I6-perl	1.04	14.8%	10.0	19.9%
N2	I6-bzip2.1	1.08	35.7%	11.6	5.6%
N3	I6-bzip2.2	1.00	24.3%	11.7	1.2%
N4	I6-sjeng	0.92	0.4%	2.8	0.7%
N5	MI-dijkstra	1.23	17.1%	18.9	0.5%
N6	MD-g721-enc	1.23	0.0%	0.0	0.0%
N7	I6-h264ref	1.07	16.7%	10.0	0.0%
N8	I6-astar	1.08	8.2%	6.7	0.0%
N9	I6-bzip2.3	0.99	0.9%	1.5	0.0%
N10	F0-art	0.47	75.0%	129.3	0.0%
N11	PB-continuous	0.82	45.9%	13.6	0.0%
N12	I0-eon	1.20	0.0%	2.6	0.0%
N13	MD-jpeg.d	1.34	0.7%	0.6	0.0%
N14	MI-rijndael	1.74	0.0%	0.6	0.0%
N15	BI-predator	1.24	0.0%	0.0	0.0%
N16	MN-bayes	1.11	0.0%	0.0	0.0%
N17	MD-adpcm.d	1.01	0.0%	0.0	0.0%
N18	MD-adpcm.e	0.86	0.0%	0.0	0.0%

Table 7: Multi-programmed workloads used in this chapter. Refer to Table 7 for individual benchmark names.

Dual-Core	
Name	**Apps**
T:N-A	T3,N11
T:N-B	T1,N5
T:N-C	T0,N3
T:N-D	T4,N7
T:N-E	T5,N7
T:N-F	T2,N9
T:T-A	T3,T1
T:T-B	T0,T4
T:T-C	T5,T2
T:T-D	T0,T3
T:T-E	T4,T6
T:T-F	T0,T2
T:T-G	T3,T4
N:N-A	N13,N14
N:N-B	N0,N12
N:N-C	N9,N8
N:N-D	N5,N11
N:N-E	N5,N0
N:N-F	N3,N12
N:N-G	N11,N2

Quad-Core	
Name	**Apps**
1T3N-A	T3,N11,N13,N18
1T3N-B	T1,N5,N14,N17
1T3N-C	T0,N0,N3,N16
1T3N-D	T4,N6,N7,N12
1T3N-E	T5,N4,N7,N15
F, G	see text
2T2N-A	T0,T2,N8,N9
2T2N-B	T0,T3,N2,N11
2T2N-C	T1,T4,N5,N10
2T2N-D	T0,T6,N8,N9
2T2N-E	T1,T2,N1,N9
2T2N-F	T1,T2,N1,N9

MediaBench and MiBench suites which are more geared toward embedded environments and tend to have smaller working sets.

5.2 Containing Thrashing Workloads

In this section, we first introduce two modifications to the conventional UCP approach to reduce its implementation complexity. These modifications reduce both the area and power costs of UCP-like cache partitioning. Both approaches still require the use of some mechanism to explicitly compute the desired cache partitioning.

5.2.1 Way Merging

Recent trends in processor design have shown a slow but steady increase in the set associativities of last-level caches. As discussed earlier, optimal partitioning requires the consideration of $O(w^N)$ possible partitionings for w ways and N cores, and even the Lookahead algorithm requires $O(w^2N)$ operations. If LLC associativities continue to increase to, say 32 or 64, this cost can become quite large. While the time required to perform the partitioning may not be a major concern because partitioning only occurs once every few million cycles, the hardware complexity and its associated design and verification costs can still quickly become prohibitive.

We make the observation that, in practice, over-allocating a few ways for a core does not have too much impact on overall performance. For example, this may occur when allocating one more way for core-i versus core-j results in a similar reduction in misses (i.e., both cores have approximately the same marginal utility for one additional cache way). To take advantage of this, we propose *Way Merging*, where we combine or merge the UMON hit information for several ways together. In a UCP cache, if core-i has a (shadow tag) hit in recency position k, then core-i increments its k^{th} UMON counter, UMON$[i][k]$ as shown in Figure 20(a). When combining m ways together with Way Merging, the same hit causes an increment in counter UMON$[i][\frac{k}{m}]$, as shown in Figure 20(b). As a result, UCP with Way Merging only uses $\frac{1}{m}$ of the total counters.

Using the reduced set of UMON counters, we partition the cache as if it only had $\frac{w}{m}$ ways using any conventional partitioning algorithm such as Lookahead (the number

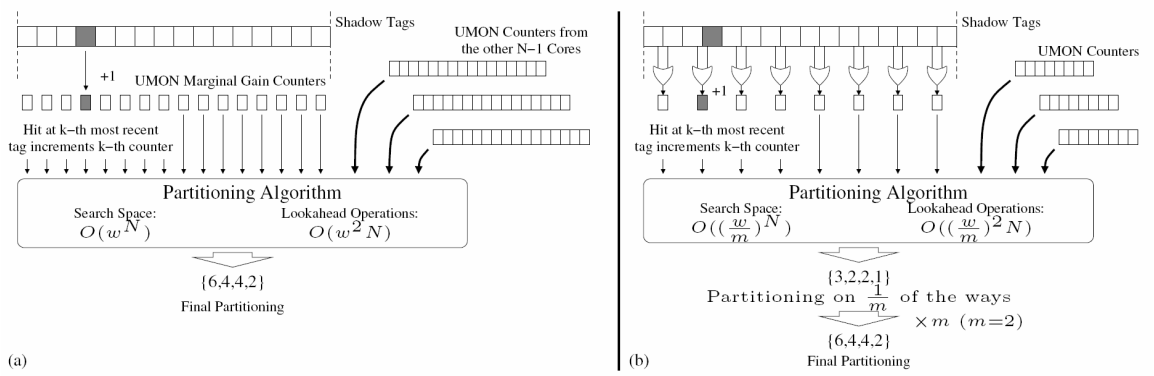

Figure 20: (a) Detail of a shadow tag hit interaction with UMON counters and partitioning logic, and (b) using Way Merging to reduce the number of UMON counters and the complexity of the partitioning logic.

of operations for the Lookahead algorithm is included in Figure 20). At the end of the partitioning process, each core $c_0, c_1, ..., c_{N-1}$ will be given an allocation $p_0, p_1, ..., p_{N-1}$, respectively, such that $\sum_{i=0}^{N-1} p_i = \frac{w}{m}$. Since the real cache has w ways in total, each core c_i then gets assigned $p_i \cdot m$ ways. While the overhead for storing the counters is very small to begin with, the savings for Way Merging really comes from the reduction in the complexity of the partitioning algorithm. Consider the case where N=4, w=16 and $m = 2$. For conventional partitioning, there are 455 possible partitionings that must be considered. With Way Merging, this search space is reduced to only 35 possible partitionings. The Lookahead algorithm now only needs to consider $O\left(\left(\frac{w}{m}\right)^2 N\right)$ possible partitionings with Way Merging; this is a reduction in complexity by a factor of m^2.

5.2.2 Thrasher Segregation

Our first approach for specifically targeting thrashing applications makes use of the UMON online-monitoring mechanism to classify programs into thrashing vs. non-thrashing groups. Each group is then treated as if it were simply one combined application with respect to partitioning. The idea is that all of the thrashing applications will be placed within one partition, and all of the other applications will be placed in a second partition. The poorly-behaving thrashing applications will only be able to negatively impact the other thrashers, whereas the better behaved applications will maintain isolation from the thrashers. This

approach leverages our earlier observation that LRU performs similar to explicit partitioning when only non-thrashing applications compete for the same cache resources. Therefore within each partition, the fact that we do not perform any further core-/sharing-aware cache management should not have any significant performance impact.

To implement this Thrasher-Segregation scheme, we actually do not need to maintain per-core marginal gain counters (as is used in UCP), but we instead replace these with one set of counters *per group*. Since the number of groups is fixed at two, the overhead of the marginal gain counters does not change even if the number of cores increases. We do, however, need to add one extra set of shadow tags per group. The idea here is that the shadow tags track what the cache's contents would be if *that group* had exclusive access to the entire cache. From this, we can update the per-group marginal gain counters. In the example in Figure 21, cores c_0 and c_2 are identified as thrashing and cores c_1 and c_3 are non-thrashing using the same shadow tags as described earlier to track the hypothetical per-core contents of an unshared cache. Figure 21 also shows the one extra set of shadow tags for tracking what the cache's contents would be if only the thrashing applications used the cache, and the second set for the non-thrashing applications, and how they each update separate sets of marginal gain counters. While this approach requires two *more* sets of shadow tags, this is a constant cost that does not increase with the number of cores. Finally, the marginal gain counters are used to partition the cache between the two groups.

In summary, the original per-core shadow tags are used to classify each core's thrashing behavior, and then the per-group shadow tags are used to update the per-group marginal gain counters. The primary benefit is that the partitioning decisions are only ever made between the two groups (if all applications belong to the same group, then the cache remains unpartitioned, or equivalently one of the partitions receives an allocation of zero ways). Note that this partitioning is far simpler than UCP for $N>2$ (i.e., more than two cores). Performing an optimal partitioning between two groups only requires considering w-1 possible partitionings, independent of the number of cores, whereas an approach that provides explicit partitions for each core needs to consider $O(w^N)$ possibilities. The advantage of Thrasher Segregation is that the partitioning complexity is fixed independent of the number

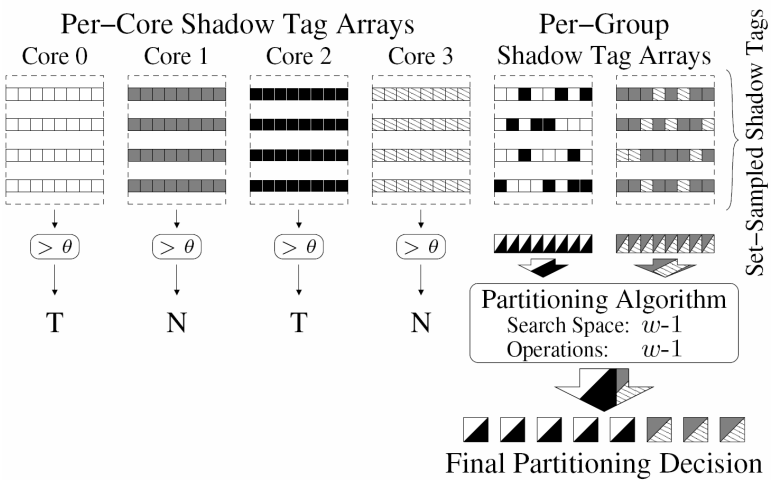

Figure 21: Shadow tag organization of Thrasher Segregation. Note that only two sets of UMON counters are needed, independent of the number of cores.

of cores. The cost is an additional two sets of shadow tags.

5.2.3 Performance

Way Merging and Thrasher Segregation (TS) both reduce the complexity of UCP-based cache management. Figure 22 shows the performance of these approaches compared to an LRU-based unmanaged cache for four-core workloads (also listed in Table 7), with sub-plots (a), (b) and (c) showing the results for the throughput, weighted speedup, and harmonic mean fairness metrics, respectively.

Across all of the 1T3N and 2T2N workloads, our proposed simplifications to UCP still provide decent performance compared to UCP. Way Merging (we set $m = 2$, i.e., every two ways are merged together) perturbs the performance only slightly, which is expected because after k ways have been allocated, the marginal gain of the $k{+}1^{\text{st}}$ way is typically not very large. On average, for the throughput, weighted speedup and harmonic mean fairness metrics, the Way Merging itself provides very close speedup to UCP, and the TS and the TS combined with Way Merging can provide about half of the gain of UCP.

There are several workloads (1T3N-C,D,E) where TS actually provides even better performance than UCP. At first, this may seem counter-intuitive that an approximation to optimal partitioning may perform better, but the optimal partitioning approach (UCP)

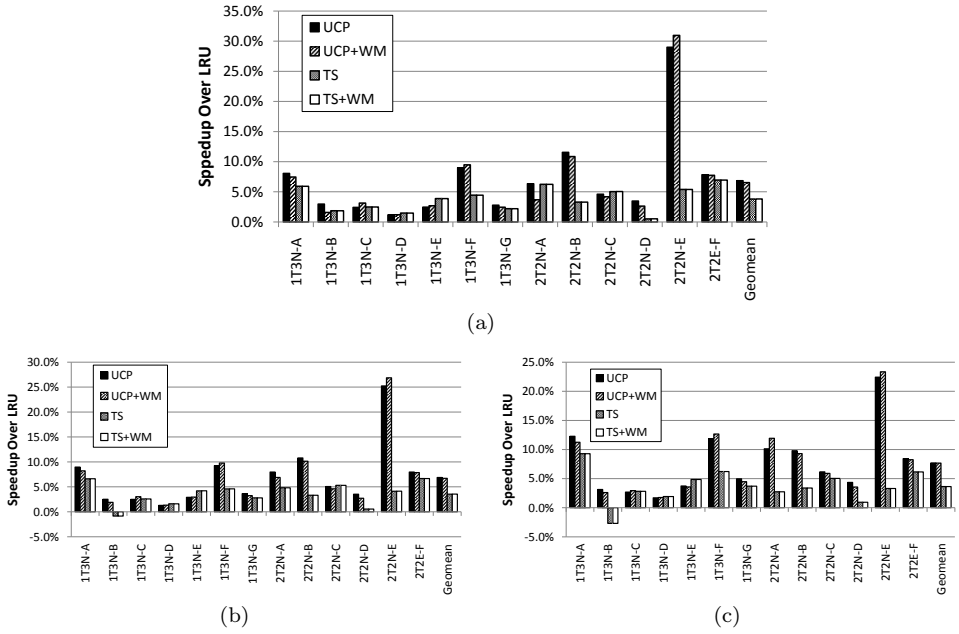

Figure 22: Performance comparisons of UCP, UCP with Way Merging, Thrasher Segregation (TS), and TS with Way Merging. All results are normalized to the performance of an unmanaged LRU cache, using the (a) total throughput, (b) weighted speedup (c) harmonic mean of weighted IPC metrics.

assumes *disjoint* partitions for each thread. In TS, all of the non-thrashing threads share the same cache space without any further enforcement. As a result, threads may "steal" capacity from other threads in the sense that at any given moment, a thread may occupy more space than it would otherwise be allowed in a strictly partitioned approach. The benefits of relaxing the strict partitioning requirement have also been demonstrated in other studies [46, 58].

5.3 Eliminating Partitioning-Complexity Dependence on Associativity

From our Thrasher Segregation technique, we observed that when thrashing workloads are present, the cache allocation for these threads was usually close to a constant factor times the number of such threads (i.e., for T thrashing applications, approximately $T \cdot c$ ways were allocated to these programs, or c ways per thrasher). Furthermore, we have observed that overall performance is not very sensitive to small changes in the number of ways allocated to thrashers; performance for $c \pm \epsilon$ ways is similar to when thrashers receive exactly c ways each. Furthermore, recall that we observed that when non-thrashing applications share the

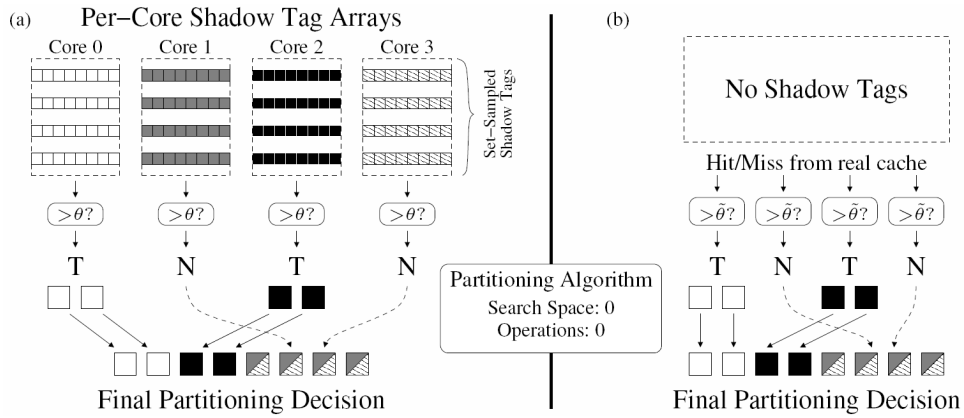

Figure 23: (a) Shadow tag, thrasher detection logic, and example partitioning for the Thrasher Caging approach, and (b) hardware changes when using Approximate Thrasher Detection.

cache, the overall performance for UCP and non-managed LRU end up being similar. We leverage these observations to further simplify the mechanisms for partitioning the cache.

With the Thrasher-Segregation scheme, there are still some situations where placing all of the thrashers into the same partitions can cause some performance degradations. While the observation that similarly-behaved applications *usually* achieve similar performance under either LRU or explicit partitioning, there are some cases where this does not hold as well. One example is if two applications both classified as exhibiting thrashing behaviors have different access frequencies or have clustered accesses. If one program accesses the cache much more frequently than the other (of course both programs access the cache a lot to begin with since they have been classified as thrashers), then it will tend to evict all of the other program's cache lines. Even though most accesses are misses, not *all* accesses are misses, and so what few potential hits could have been achieved by the second program are obliterated by the higher access rate of the first. With alternating/bursty access patterns, a similar situation can occur where a quick burst of accesses from one core quickly evicts all of the lines of the other core.

5.3.1 Thrasher Caging

Our second approach improves on the Thrasher-Segregation scheme by providing per-thrasher partitions, or *cages*. A cage is simply a special case of a partition with a fixed,

predetermined size. To keep the mechanism simple, each thrashing application is quarantined within its own cage consisting of c ways.[1] The cage size c is statically fixed for all programs/workloads. We have empirically chosen a cage size $c = 2$, but have found that for the thrashing applications, slight changes in the cage size do not have very much performance impact. For truly thrashing applications, they will continue to just access and miss in the cache but will only do so within their own cage. For a thrashing application that shows some or occasional locality in its reference patterns, it will be able to generate more hits as these lines can be kept in its own cage, but it continues to *not* affect the other programs during the more frequent thrashing periods. The non-thrashing applications share whatever left-over ways are available in an otherwise unmanaged (LRU) fashion. Figure 23(a) shows the hardware structures and the final partitioning for four cores, where cores c_0 and c_2 are thrashing.[2]

From an implementation perspective, this Caging approach is much more lightweight than the UCP approach. The complex partitioning mechanism can effectively be *completely eliminated* as the partition sizes are a fixed function of the programs' thrashing classifications. As a result, all of the UMON counters can also be eliminated, too. The only significant remaining overheads are the per-core shadow tags used for classifying whether a program exhibits thrashing behavior. Note that the partitioning mechanism is where most of the complexity lies when the number of cores or the set-associativity increases. Thrasher Caging reduces the number of operations from $O(w^2N)$ (for Lookahead) to effectively zero regardless of the number of cores or the cache's set associativity.

5.3.2 Approximate Thrasher Detection (ATD)

The Thrasher Caging approach's only substantial remaining overhead is from the per-core shadow tags used for the Thrasher classification. Note also, that the only role served by the shadow tags for Thrasher Caging is to identify when programs exhibit thrashing behaviors.

[1] In the case that all N programs are thrashers, then each core is allocated $\frac{w}{N}$ ways, which may be larger than c.

[2] The example partitioning in Figure 23 shows the caged partitions on the left and the shared partition on the right, but this is merely for illustrative purposes. The specific cache ways used by individual cores need not be physically contiguous.

One would suspect that the fine-grained per-way marginal utility-tracking capabilities of the per-core shadow tags is an overkill. This is in fact the case, and we describe a simple alternative to approximate this information, which we call Approximate Thrasher Detection (ATD).

Our approach is simple: we only track the absolute number of misses such that if a core causes more than $\tilde{\theta}_{miss}$ misses, then the core is considered to be thrashing.[3] Considering only misses without considering hits could potentially lead to cases where an application is unfairly punished (i.e., it has a high average hit rate over many memory accesses, but it still results in more than $\tilde{\theta}_{miss}$ misses). Our intuition is that counting only misses should still work for the *aggregate* system performance (as measured by, for example, overall throughput or weighted speedup) because whatever benefit those misses provide for the one application, the remaining $> \tilde{\theta}_{miss}$ misses would still wreak havoc for the other non-thrashing programs. The selection of the exact values for these thresholds are discussed in the analysis section. We also considered a version where we use the miss *rate* rather than the absolute number of misses, but it turns out that tracking only misses performs better while being easier to implement.

Note that for our ATD, we only track the miss statistics on the actual misses observed on the real cache contents, independent of whether these accesses would have been hits in an unshared cache. The intuition for why this is still accurate is that for a thrashing workload, whether it receives a few ways or the entire cache, the majority of its accesses will be misses and therefore the number of misses observed in the real cache or an unshared cache will still be very similar (i.e., providing the entire cache for this application still will not significantly increase the number of hits). ATD completely removes *all* shadow tags, rendering the total storage overhead for our simplified partitioning scheme to only one counter per core to track per-core misses. Figure 23(b) illustrates the final design of Thrasher Caging with ATD.

[3]We use the notation $\tilde{\theta}$ instead of θ to emphasize that this threshold corresponds to an approximation of the previous classification approach.

5.3.3 Performance of TC and ATD

We evaluated Thrasher Caging (TC) on a variety of four-core workloads listed in Table 7. We simulated workloads with 4T0N (four thrashing programs, no non-thrashing), 3T1N, 2T2N, 1T3N, and 0T4N. Figure 24 only shows the results for 1T3N and 2T2N; the other workloads showed very little benefit from the baseline UCP, and TC also does not cause any performance problems for these workloads, so they are omitted for brevity. We also considered dual-core 1T1N applications with similar results [57]. Figure 24 shows the performance of these approaches compared to an LRU-based unmanaged cache for four-core workloads, for all the three metrics. While TC was proposed to simplify/eliminate the complex partitioning decision logic, it provides 10.6% performance improvement on average which is better than TS's 3.5%. The reason that TC performs better than TS is that even though the thrashers have been isolated from the non-thrashers, the thrashers themselves can still occasionally cause performance problems within their group. TS addresses the first problem of protecting the non-thrashers, and TC addresses the secondary problem of protecting thrashers from each other.

Figure 24 also includes the performance results for TADIP. Across our simulated workloads, TADIP performs slightly better than UCP (with a lower implementation overhead). On average, our TC approach performs better than both UCP and TADIP, although there are individual workloads where UCP or TADIP is the best approach. Only for the fair speedup metric does TC not perform as strongly as the other approaches, but it still achieves fair speedup results close to the others and significantly better than an unmanaged LRU cache.

TADIP and TC actually provide similar benefits in different guises. When a thrashing application is present, TADIP effectively isolates this thread by forcing the thread's cache lines to be inserted at the LRU position. The non-thrashing threads will be inserted at the MRU position, and as a result, the overall scheme behaves similar to thrasher caging where the cage size is one, and all thrashing programs share the same cage. There are a few scenarios where TADIP's approach may break down. First, TADIP does not perform strict LRU insertion, but rather performs a probabilistic insertion where MRU insertion

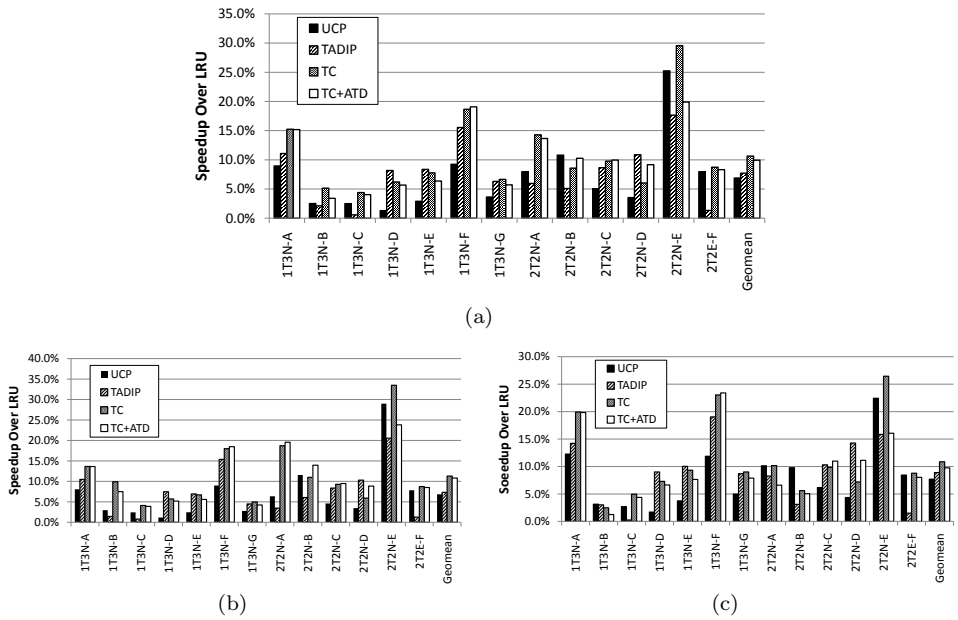

Figure 24: Performance comparisons of Thrasher Caging (TC) and TC with Approximate Thrasher Detection. All results are speedups over an unmanaged LRU cache, using the (a) weighted speedup, (b) IPC throughput and (c) harmonic mean of weighted IPC metrics.

occurs with a probability $p = \frac{1}{32}$ and LRU insertion otherwise. For an application with extreme thrashing that inserts lines into the cache at a very high rate relative to the access frequencies of other cores, even the one out of 32 lines being inserted at the MRU position is enough to cause many other lines to be evicted. Second, there are some cases where maintaining some level of isolation, even between thrashing applications, is still beneficial. For example, many thrashing applications simply stream through memory in a sequential access pattern. For such programs, hardware prefetchers can easily predict the pattern and prefetch data into the cache. If these are inserted at the LRU position, however, the prefetched lines may be evicted before the corresponding core even has a chance to make use of the line. With separate per-thrasher partitions, TC avoids this situation. This may be part of the reason why the relative benefits of TADIP are reduced in a environment where prefetching is enabled [27].

Figure 24 includes the performance of TC with approximate thrasher detection (ATD) using the number of absolute misses. For the majority of benchmarks, this metric serves as a more accurate proxy for thrasher detection than the relative miss rate. This is not entirely

surprising since a high miss rate could result from only a small number of accesses (e.g., in the extreme case, one miss on one access would result in a 100% miss rate). In either case, TC+ATD (15.9% and 17.0% better than LRU on the weighted speedup metric for $\tilde{\theta}_{MissRate}$ and $\tilde{\theta}_{miss}$, respectively) performs better than UCP (13.5% over LRU), and comes fairly close to TC without ATD (18.4%). There are a few individual workloads where the ATD approach actually performs better than shadow-tag-based TC. The reason for this is that the thrasher-classification criteria is itself a heuristic where the best threshold for thrasher classification will vary from one workload to the next (but we use a fixed threshold for all workloads). The "error" introduced by ATD could in fact push the effective thrasher classification to more closely mimic the classification results that would occur for a better selection of the threshold *for that workload*.

5.3.4 Implementation Cost of TC and ATD

The performance results in the previous section demonstrate that the benefits of UCP for managing a shared cache can be obtained with a hardware implementation that is much simpler and scalable. Table 8 lists the storage overheads required to implement different cache management schemes. In particular, note that for most of the approaches, the storage overhead is measured in kilobytes (KB), whereas for TC+ATD, the storage overhead is only a few *bytes*. It is also important to point out that the overheads in Table 8 do not account for the logic and state required to implement the partition-decision logic (e.g., the Lookahead algorithm) where necessary, i.e., UCP. While TADIP's storage overhead is the same as TC+ATD's, our proposed approach appears to perform slightly better according to our simulations.

5.4 Scaling and Sensitivity Analysis

Figure 25(a) shows the weighted speedups for 8-core configurations using an 8MB, 32-way LLC (the other metrics show similar trends and are omitted for brevity). The workloads feature different mixes of the same thrashing and non-thrashing applications from Table 6, although the specific workload compositions are omitted due to space constraints. The overall results are similar to the four-core results presented earlier in that TC provides some

Table 8: Summary of overheads for different cache management schemes. Example storage overhead assumes s=4096 sets, w=16 ways, N=4 cores, t=36 bits per shadow tag entry, $\alpha=\frac{1}{128}$ (DSS sampling rate), m=2 (Way Merging rate), UMON counters, ATD miss counters and TADIP PSEL counters are b=10 bits each.

	Shadow Tag Storage	Counters (UMON/miss ctrs/PSEL)	Search Space Size	Storage 4MB/16-way L2, 4 cores
UCP (*no DSS*)	$swtN$	wNb	$O(w^N)$	1.1 MB
UCP (*w/ DSS*)	$\alpha swtN$	wNb	$O(w^N)$	9.1 KB
Thr. Caging (*w/ DSS*)	$\alpha swtN$	0	0	9.0 KB
TADIP	0	Nb	0	5.0 B
TC+ATD	0	Nb	0	5.0 B

performance gain primarily due to allowing non-thrashing applications to share the same partition. In these workloads, the ATD approach introduces more performance degradation than before. It is important to note that we have *not* re-optimized the $\tilde{\theta}_{miss}$ threshold for these simulations (i.e., this uses the threshold optimized for the four-core case). Overall, Thrasher Caging is an effective approach to managing a shared cache among many cores. With ATD, TC can on average still provide the performance benefits of UCP but with a trivial hardware overhead.

Our results thus far have shown that Thrasher Caging with ATD works well for 4 and 8 cores on a processor with a shared 8MB, 32-way LLC. This cache configuration may be somewhat aggressive compared to current processors, so we also present results with 8MB/16-way and 4MB/16-way LLC's. Figure 25(b) shows the weighted speedup results for the four-core workloads. The overall results are similar to the earlier 8MB/32-way results, showing that our approach is also effective for less aggressive cache organizations.

Our Thrasher Caging approach makes use of a few parameters that need to be tuned. In particular, the size of the per-thrasher cage and the various thrasher-detection thresholds all need to be chosen appropriately. Figure 25(c) shows the weighted speedup of TC (without ATD) for various cage sizes, along with the performance of UCP for reference. While we have used a cage size of c=2 throughout this chapter, choosing a cage size of three or four does not have much impact on per-workload and overall performance. For a few workloads, having a cage too small (c=1) or too large (c=6) does adversely affect the performance.

The original thrasher classification criteria uses two thresholds: θ_{miss} and $\theta_{MissRate}$. For our four-core workloads with a 8MB/32-way cache, we found that the best values for these thresholds were θ_{miss}=100 and $\theta_{MissRate}$=0.5% (accounting for DSS). While these values may seem low, we found that for this cache size, program behaviors were very bimodal in that they either exhibited many misses or very few misses, but seldom had behaviors in between. Note also that this is a dynamic metric in that we collect these based on the number of *cycles* of execution rather than the number of instructions executed. That means a program could have a high MPKI rate, but a low IPC rate could still result in few observed misses within a fixed time interval. We experimented with a wide range of threshold values, and even using θ_{miss}=4000 and $\theta_{MissRate}$=6.0% we achieved average weighted speedups within 1.8% of those achieved with the best threshold values. So while the thresholds might be viewed as somewhat arbitrary, the performance results are not very sensitive to the exact choices.

For the approximate thrasher detection threshold $\tilde{\theta}_{miss}$ we used a value of 2000 misses. Changing the threshold by ±1000 results in less than 2.4% loss in the performance benefit over LRU. Overall, the proposed technique does not exhibit any exceptional negative sensitivity to the exact threshold value.

In this work, we have exploited the fact that cache sharing problems are generally caused by a few applications that generate a large number of misses that end up displacing the cachelines used by the other programs. By simply containing and controlling these few programs, our Thrasher Caging technique can achieve better performance than UCP with a simpler implementation, and using Approximate Thrasher Detection we can *completely* eliminate all of the shadow tag, utility monitor and partitioning logic overheads. Finding simple, low-overhead mechanisms is critical for the adoption of such techniques in more constrained embedded multi-core processor designs.

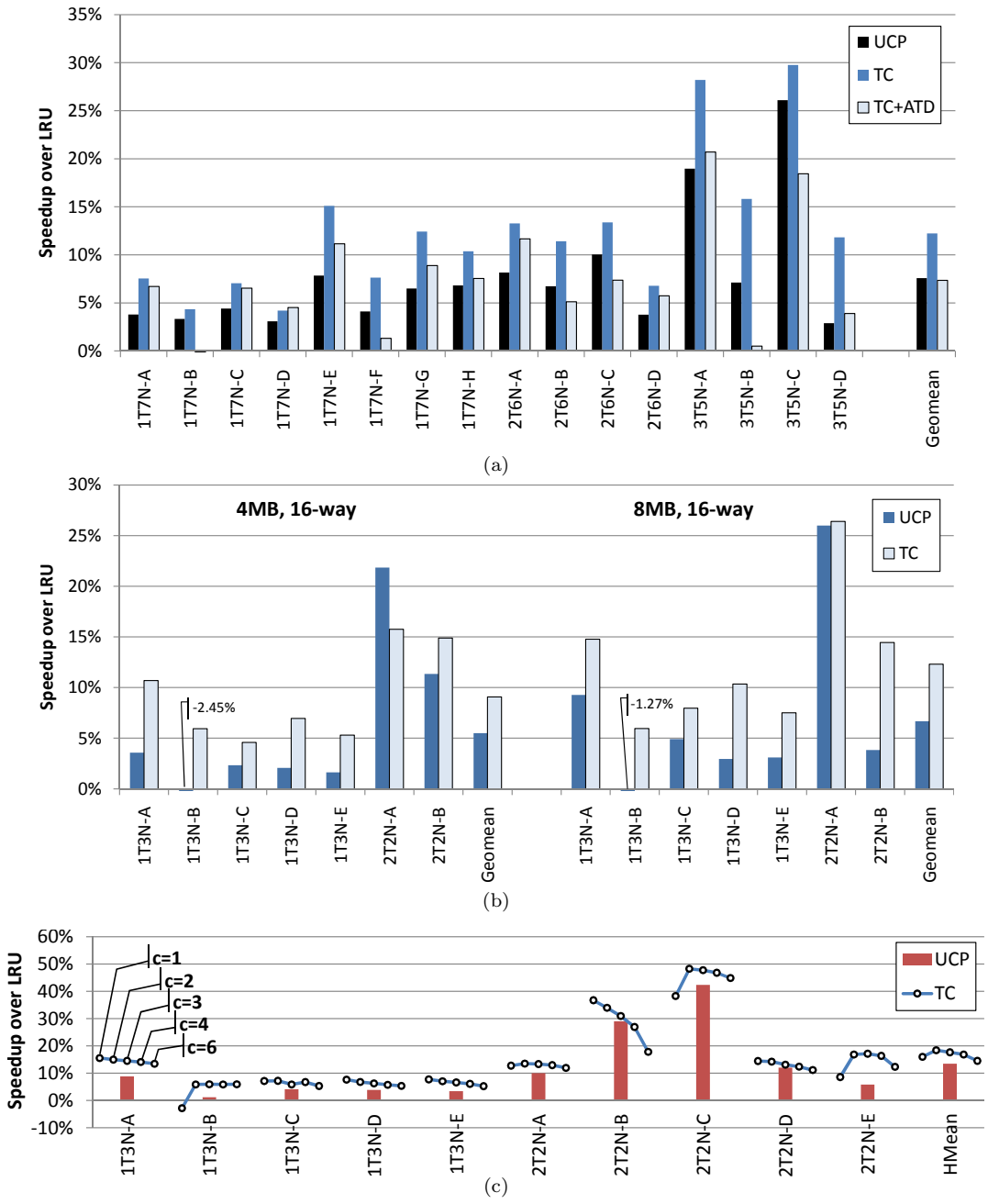

Figure 25: (a) Weighted speedup results for 8-core workloads, (b) Weighted speedup results for smaller and lower-associativity caches, (c) Thrasher Caging performance for different cage sizes.

CHAPTER VI

CACHE PARTITIONING FOR PSEUDO-LRU REPLACEMENT

The regular LRU replacement algorithm or "ture LRU" is widely studied in academic research, but in industry, modern processor designers could choose a more practical implementation of the replacement algorithm which is a pseudo-LRU(pLRU). In this chapter, we study the tree-based pLRU variation, and propose a mechanism to get an approximation of the LRU hit position for application cache utility estimate purpose. The low overhead approximation turns out to be able to provide enough accuracy to partition the cache for improving the system performance. As a result, this chapter futher supports the thesis because we present an efficient cache management scheme that is applicable to more practical processor implementations.

6.1 LRU Replacement and Approximations

An optimal replacement policy for a cache is impossible to implement because it requires knowledge of future access patterns. As a result, a variety of replacement heuristics have been proposed, of which replacing the least-recently-used (LRU) cache line has been show to perform well in practice. The LRU policy exhibits a property called the LRU stack *inclusion property* [38]. For the same number of sets, all of the contents of a v-way set-associative cache can always be found in a w-way set-associative cache, for $v < w$. By arranging the w lines of a cache set in their recency order (from most recently accessed to least), the first v elements of this *LRU stack* contain exactly the elements that would be found in the corresponding set of a v-way set-associative cache. Exploiting this property allows one to use a w-way LRU cache to simulate all possible levels of set associativity less than w.

While the LRU replacement policy has been experimentally shown to work very well in most situations, it becomes increasingly difficult to implement for caches with high set associativities. To implement LRU, every cache line must be augmented with a $\lceil \log_2 w \rceil$-bit counter that records the line's place in the overall LRU stack. Figure 26(a) shows a

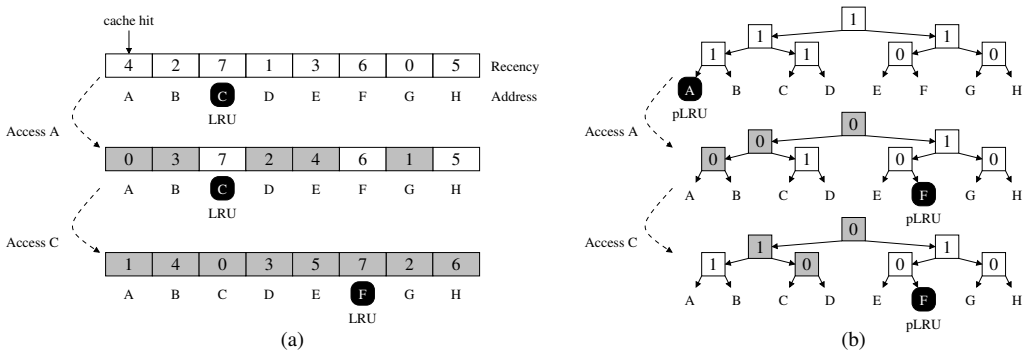

Figure 26: Example replacement policy updates for (a) true LRU and (b) pseudo-LRU policies.

simple example where accessing a single cache line causes five of the counters to be updated. The update logic must check the age of the accessed cache line, locate all other lines that have a lower position in the LRU stack, and then increment their corresponding counters (shaded). An access to the LRU position forces updates to *all* counters. The overhead for the replacement state is $O(w \log w)$ since each of the w ways requires an $O(\log w)$-bit counter.

To approximate the benefits of an LRU replacement policy while maintaining lower overheads and implementation complexities, researchers have proposed a variety of LRU approximations or *pseudo-LRU* replacement policies (e.g., NMRU, pLRU). The basic pseudo-LRU (pLRU) algorithm uses a binary-tree data structure, shown in Figure 26(b). Each node in the tree contains a single bit that encodes whether a line in the left sub-tree (0) or the right sub-tree (1) was more recently accessed. On a cache hit, the update policy only needs to update the $\lceil \log_2 w \rceil$ bits from the leaf up to the root node. To find the (pseudo-)least recently used line, we can simply reverse the process starting from the root node (easily accomplished by inverting all of the bits). By comparing Figures 26(a) and (b), one can see that the pseudo-LRU cache line is sometimes, but not always the same as the true LRU line.

The total number of bits required to store the pLRU data structure is simply w-1 per *set*, a factor of $O(\log w)$ less than true LRU. On a cache hit, updating the tree only requires setting the $\log w$ bits from the cache line up to the root node to indicate the location of

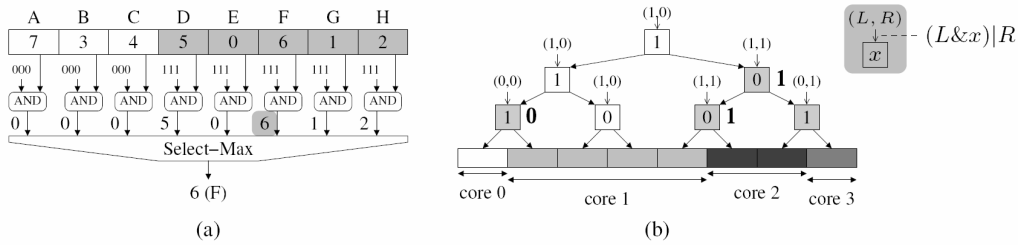

Figure 27: Selecting an replacement victim in a way-partitioned cache assuming (a) LRU and (b) pseudo-LRU replacement policies.

the most recently used line. This can be performed independent of the values of the bits in any of the other sub-trees; this is in stark contrast to the true-LRU policy that requires comparing all counters in the entire set to determine which counters need to be modified. The combination of low storage overhead and simple update rules make pLRU practical replacement policy even for highly set-associative caches.

6.2 Practical UCP for a pLRU Cache

6.2.1 Choosing a Victim

To find the evictee in a way-partitioned cache with a *true* LRU replacement policy, one can simply mask out the recency counters for lines belonging to other cores. For the example in Figure 27(a), the first three ways belong to core 0 and the remaining five ways belong to core 1. To choose a victim for core 1, we simply AND all of the counters with a mask that causes all of core 0's counters to be zeroed out (indicating MRU). Out of the remaining counters, line F has the highest tag and gets evicted. Note that this complicates the search for a victim because in a traditional (non-partitioned) LRU cache, the single unique entry with a timestamp of 7 (or w-1 in general) is the evictee. Now we must perform a "MAX" operation involving the comparison of all counters.

In a pLRU cache, the ways under the recency tree can still be partitioned into contiguous intervals, shown in Figure 27(b), where each core can only evict lines belonging to its interval. Without considering the partitioning, pLRU would normally lead you to choose core 0's cache line as the evictee (recall that '0' means the left side contains the most recently used line, so to find the pLRU line we interpret the bits in reverse: the '1' at the

root means we should go left to find the pLRU line). To account for partitioning, we must "steer" the tree traversal in such a way that it always ends up at a line within the core's allocated interval. For example, core 1 has been assigned ways 1 through 4. When trying to find the pLRU cache line, anytime we reach a node where one sub-tree does not contain *any* of core 1's lines (these nodes are indicated with shading), then we need to over-ride the node's value to force the traversal to the correct sub-tree. For example, the left-most node's left sub-tree does not contain any lines belonging to core 1.

To enforce this partition-aware routing through the pLRU tree, we simply need to generate two vectors $\overrightarrow{L}$ and $\overrightarrow{R}$. There is a one L and one R bit associated with each node in the pLRU tree (not stored with each bit, though; each core maintains only one global $\overrightarrow{L}$ and $\overrightarrow{R}$ vector). Figure 27(b) shows the respective bits as (L, R) over each node. For each node, we simply take the node's pLRU bit and AND it with its L bit. If the L bit is a zero, then the outcome will be zero which in turn forces the routing into the right sub-tree. Similarly, we take the pLRU bit and OR it with its R bit; if the R bit is a one, then that will force the routing into the left sub-tree. As a result, performing partition-aware pLRU victim selection is identical to the original pLRU algorithm, except that we need to first perform two simple bitwise logical operations on the pLRU state. The $\overrightarrow{R}$ and $\overrightarrow{L}$ bit-vectors are small (w-1 bits per vector) and can be easily pre-computed during the partitioning process.

On a cache hit, we simply update the pLRU state in a partition-oblivious manner. While this can potentially cause cross-partition pollution of the pLRU state, we observed that this had little impact on overall performance.

6.2.2 Approximating Utility from pLRU State

The UCP technique uses a set of shadow tags to emulate the state of the cache if one core had sole use. In a true LRU cache, a hit in position i of the recency stack (as determined by the shadow tags) causes the UMON to increment $H[i]$. In a pLRU cache, we can also provide shadow tags, but with the pLRU recency tree, it is less clear how a hit at a particular cache line corresponds to the true LRU recency position. We propose to estimate or guess a

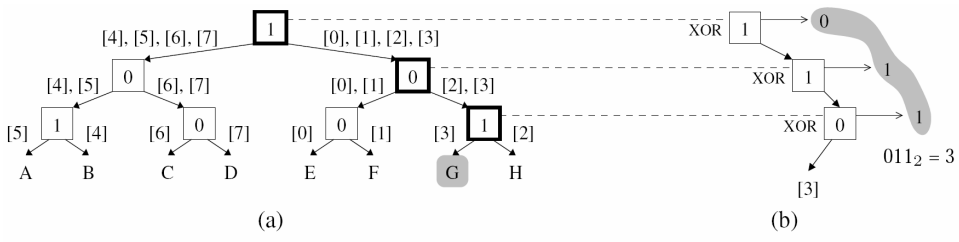

Figure 28: (a) Recursive refinement of assumed recency positions of cache lines, and (b) a direct method of computing the pseudo-recency position of a cache line.

pseudo-recency position (pRP) from the incomplete information in the pLRU tree. Suppose a node's pLRU bit is '1', which indicates that its right sub-tree is more recently used than its left sub-tree. A simple (but sometimes inaccurate) assumption one can make is that *all* of the nodes in the right sub-tree are more recently used than the nodes in the left sub-tree. Suppose we had a hit on line G as shown in Figure 28(a). We start at the root of the tree, where the value of one indicates that the right sub-tree is more recently used, and therefore contains lines [0], [1], [2] and [3], where [i] denotes the cache line is believed to be at the i^{th} recency position. At the next level, the pLRU bit is zero meaning that the left side is more recently used. We therefore assume that [0] and [1] are on the left, and [2] and [3] are on right. Finally, the last bit on our way to G is one, and so we estimate G as being a hit in the third recency position causing an increment to $H[3]$.

To quickly compute the pRP, we can actually just take the pLRU bits on the path to the line where we had the hit (these are shown with bold boxes), and then we also consider what the pLRU bits would be if the line was most recently used. Figure 28(b) shows the values of the bits to mark G as MRU. Now, by simply taking a bit-wise exclusive-OR of these two sets of bits, we get 011_2 which is equal to G's pRP (3).

In this pLRU approximation, the In-Cache UMON (ICEmon) idea introduced before can also be integrated in here. In the performance evaluation section, we can see that the ICEmon approximation is also working well with the pLRU approximation.

Table 9: SPEC2006 benchmarks used in our workload creation. Applications with multiple inputs are differentiated with a hyphenated abbreviation of the input filename.

Sensitive (S)	astar-r, bzip2-c, bzip2-l, h264ref-f, hmmer-n, perl-s
Thrashing (T)	libquantum, mcf, omnetpp, gcc-g23, lbm, milc, soplex-p, soplex-r, leslie3d, sphinx3
Neutral (N)	bzip2-p, go-13, go-n, go-tc, go-td, h264ref-s, hmmer-r, perl-d, sjeng, gcc-s04, gromacs, zeusmp

6.3 Experimental Evaluation

We classified each of the SPEC programs into one of three groups as shown in Table 9. The thrashing 'T' workloads access the cache very frequently and tend to cause other cores' cache lines to be rapidly evicted. The sensitive 'S' workloads can be easily hurt by sharing the cache with other poorly behaved applications (i.e., the thrashing programs). The neutral 'N' benchmarks have very high hit rates, but only require a small number of ways. From these groups, we create several workloads. For the dual-core scenarios, we mostly focus on combinations of S and T where effective cache partitioning is critical. We also evaluate several other types of workloads, but these are just for ensuring that our proposed techniques do not unintentionally cause problems when partitioning is not required.

Figure 29(a)-(c) shows the performance results for LRU, pLRU and several partitioning schemes. All of the partitioning results make use of a full set of shadow tags per core. This allows us to isolate the impact of coping with a pLRU cache; we will deal with sampling issues shortly. First, an unpartitioned cache using pLRU sometimes does better, and sometimes worse, than a true-LRU cache, but overall (geometric mean across the T:S workloads), pLRU suffers a 5.4% performance loss compared to LRU.

When we apply UCP to LRU with a one-million cycle repartitioning interval, we observe trends similar to previous studies, with a geometric mean weighted speedup improvement of 15.5% [45]. When we apply UCP to a pLRU cache using our pseudo-recency estimation technique (pRP), we also see similar gains of +13.2% relative to the baseline unpartitioned pLRU cache. To observe the effect of the pRP technique, we also considered a hypothetical cache where the main cache uses a pLRU policy, but the shadow tags make use of true LRU for determining the utility vectors; this configuration is denoted as P/L (pLRU cache/LRU

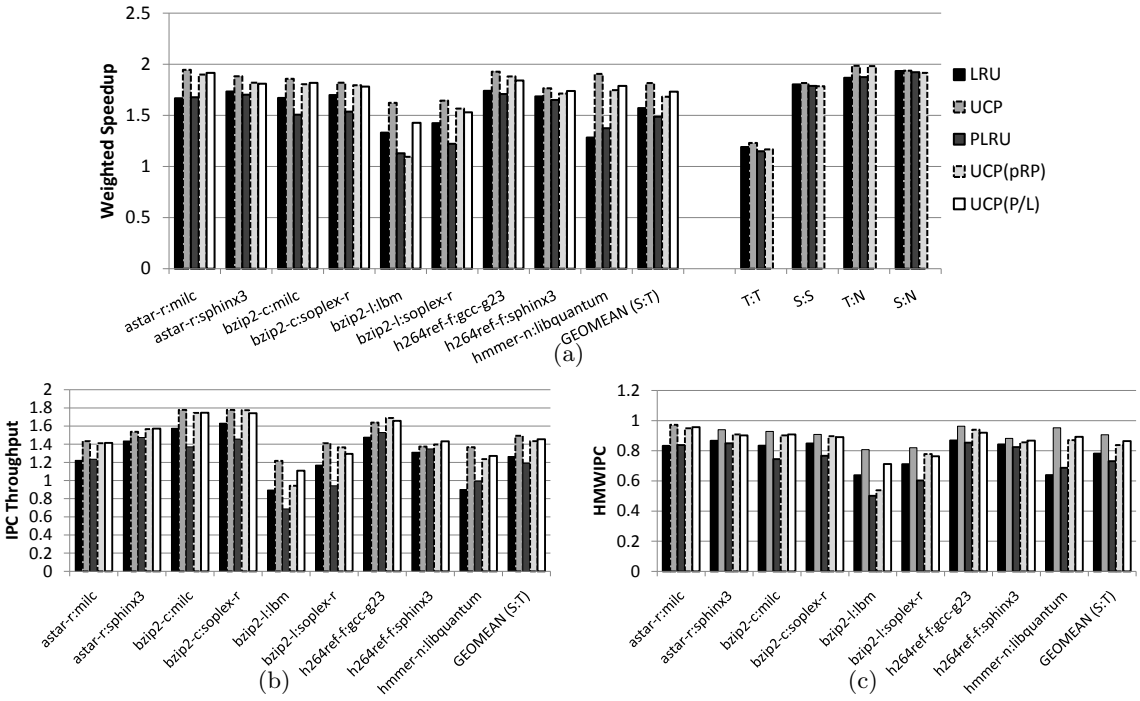

Figure 29: (a) Weighted speedup, (b) IPC throughput, and (c) harmonic mean of weighted IPCs (HMWIPC) for unpartitioned and partitioned caches assuming full shadow tags.

UMON) in Figure 29. In this fashion, the partitioning decisions are based on the true LRU recency information, but the replacement decisions within the main cache are still based on pLRU. The performance increase over the baseline pLRU cache is +16.6%, which shows that our pRP estimation (13.2%) is not losing too much information despite being a fairly coarse approximation.

We also simulated all of these configurations with other benchmark mixes. For these other groups, the per-workload results did not exhibit much variance, and so we simply provide the per-group geometric means. Overall, partitioning and even the baseline replacement policy do not have a large impact on performance for these other workloads.

The preceding partitioning results all make use of full shadow tags (one set per core without any sampling) to focus on the impact of using pLRU replacement and pLRU-based estimation of pRP. Figure 30(a)-(c) shows the performance of the baseline unpartitioned cache using pLRU, a partitioned configuration using a full set of shadow tags (identical

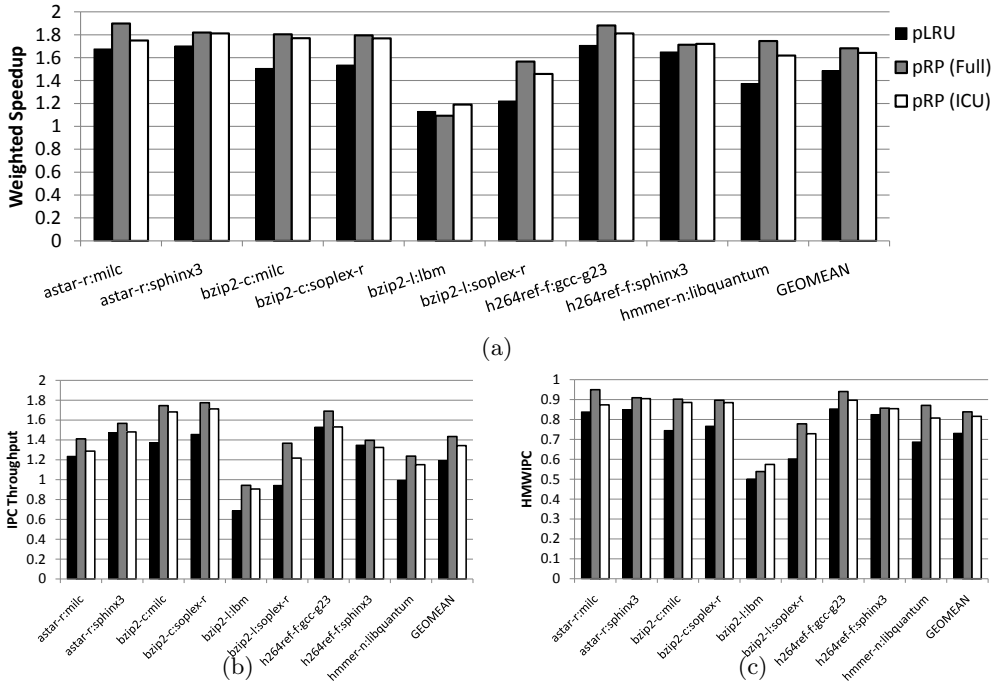

Figure 30: (a) Weighted speedup, (b) IPC throughput, and (c) harmonic mean of weighted IPCs (HMWIPC) for the baseline unpartitioned pLRU, UCP using pRP and full shadow tags, and UCP using pRP and ICU.

data as UCP(pRP) from Figure 29), and our In-Cache UMON (ICU) technique. Our 4MB L2 cache has 4096 sets, and for the partitioned configurations, we provide 16 samples (or leader sets) per core. For ICU, we reserve r=5 ways for the non-monitored cores in each leader set.

As mentioned earlier, UCP using full shadow tags and pRP achieves a 13.2% improvement over an unmanaged pLRU cache; the results in Figure 30(a) show that our ICU approach still provides a +10.6% improvement over plain pLRU. Recall that these results apply to a practical pLRU-based cache, and that the ICU approach has completely eliminated the overhead for the shadow tags.

The trends for the other metrics are very similar. Geometric mean throughput for pRP(ICU) is 12.7% higher compared to the unpartitioned pLRU baseline. Similarly, the harmonic mean of weighted IPCs (HMWIPC) shows a 11.6% improvement over the baseline. The pRP(ICU) partitioning approach provides a solid improvement on both of the performance metrics. The strong results on the HMWIPC metric indicate that pRP(ICU)

also maintains fairness, as none of the individual workloads suffers from a particularly large reduction in IPC rates.

We experimentally determined to reserve r=5 ways for the non-monitored core when using ICU. We ran simulations sweeping $r \in \{1..7\}$ and simply chose the best setting. For a dual-core processor, reserving more than seven ways for a 16-way cache does not make sense as this would result in hit counters that never register any hits beyond eight ways, which in turn would always cause the partitioning algorithm to assign an even 50-50 partition. Having even five ways reserved means that the leader sets may suffer from additional misses because they can never be repartitioned. This did not prove to have much impact on overall performance. We first evaluated a modified ICU algorithm where the leader sets slowly rotate through the cache. This way if any single set is very popular, the impact of the non-repartitionable leader set can only have a negative impact for a short duration. Similarly, we also experimented with temporal sampling, where we only collect utility information for a brief sampling period at the start of the one million cycle interval, and then the leader sets stop being leaders and the entire cache gets repartitioned based on the information collected during the sampling period. Neither of these approaches had a substantial impact on performance, and so we chose to take the simpler approach.

CHAPTER VII

THREAD-AWARE DYNAMIC SHARED CACHE COMPRESSION

The previous chapters study how to manage a fixed size cache between cores to improve the performance. Researchers found cache compression an approach to effectively enlarge the cache space and therefore alleviate the capacity competition problem. Previous studies showed simple and low latency cache compression/decompression engine can be implemented based on the existing cache architecture. This chapter proposes a low overhead compression benefit tracking technique, which gives smart compression decisions on a per-thread basis. We also consider the interaction between the cache compression and the cache management schemes. This chapter will show that the cache management and the cache compression can work nicely together to improve the system performance by using low overhead hardware and therefore the thesis is further supported.

7.1 Introduction

The last level cache plays an important role in modern processor design, because the off-chip memory access usually has a very long latency. While increasing the cache size can reduce cache miss rates, larger structure results in longer access latencies, more area and higher power. Previous studies showed that the data stored in the last level cache is highly compressible. This makes cache compression an attractive approach to reduce capacity misses, because it increases the effective cache size. Figure 31 shows an example structure of a multicore processor with a shared cache that supports compression.

Though compression can help to reduce capacity misses, it unfortunately also increases access latencies because cache lines stored in compressed formats have to be decompressed before being used. One needs to consider whether compression is even beneficial before enabling compression. If a program has a small working set, most of the accesses are cache hits. In this case, compression does not help too much for reducing misses, but the decompression latency will hurt performance. If the compression can reduce a significant

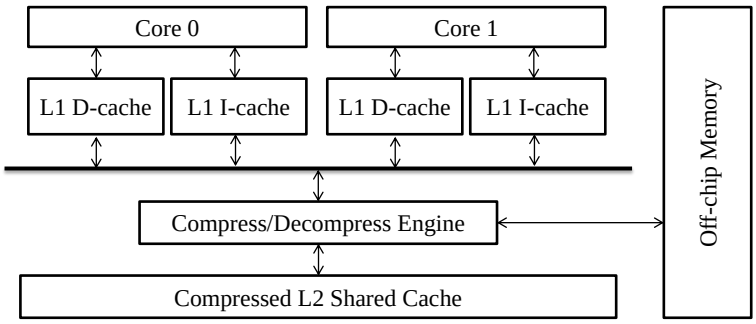

Figure 31: An example cache hierachy design where the L2 cache is compressed by a 5-stage pipeline compressing engine.

amount of misses, it is likely going to help performance since a last-level cache miss may be hundreds of cycles.

The cache miss penalty is very expensive compared to the decompression latency. Let the cache hit latency be h, the miss penalty be m, and the decompression latency be d. Suppose the number of hits and misses are H and M. After compression δ misses are converted into hits. The total access latency without compression would be

$$t_{NoCompress} = Hh + Mm$$

And the access latency with compression is

$$t_{Compress} = (H + \delta)(h + d) + (M - \delta)m$$

The compression is beneficial when

$$t_{NoCompress} > t_{Compress}$$

We can then get

$$Hh + Mm > (H + \delta)(h + d) + (M - \delta)m$$

or

$$\frac{\delta}{H} > \frac{d}{m - (h + d)}$$

Because h and d are very small compared to m, we can approximate the inequality as

$$\frac{\delta}{H} > \frac{d}{m}$$

We observed that the decompression latency divided by the miss penalty d/m is a property of the system. The number of the relative increase in cache hits is a property of the workloads. Compression is helpful when this inequality holds. In general, it is not always possible to know the relative increase in cache hits due to compression ahead of time to make a fixed decision. Instead, a system should dynamically observe the program behavior and adaptively make compression decisions. This problem becomes more interesting in a multicore shared cache scenario, because different applications exhibit different reactions to compression, and the benefit of compression is also impacted by how the applications interact in the cache. In a multicore environment, it is much less clear how to make a compression decision.

7.2 Related Work

Alameldeen and Wood proposed dynamic cache compression based on a decoupled variable-segment cache structure [1]. The compression algorithm they use is Frequent Pattern Compression (FPC) [2]. In this design, the cache data storage is broken into segments. Different cache blocks can take up different amounts of storage depending on compression ratios. Extra logic is added to deal with locating the desired block. An adaptive approach is also proposed to dynamically decide whether to enable compression so that the data are compressed only when necessary. Our work uses this decoupled variable-segment cache structure with FPC as a baseline. The variable-segment structure is based on a regular cache architecture and it can take the most benefit from the compression because it does not have restrictions such as compression line pair in [10]. FPC algorithm has lower latency (5-cycle), less hardware complexity and comparable compression rate compared to dictionary based compression algorithms.

Hallnor and Reinhardt proposed a unified compression scheme which is based on the indirect-indexing cache (IIC) structure [20, 22]. In an IIC design, the tag is not associated

with a single tag entry, but it contains a pointer to one data block in data array. To enable cache compression, the tag is augmented to be associated with multiple data subblock pointers. A cache block is assigned only as many subblocks as are needed. Their compression algorithm is combined with the generational replacement policy. While our proposals can be easily adapted to IIC's compression, we do not quantitatively compare to IIC because the IIC needs significant changes over typical cache structures.

Chen et al. proposed C-pack cache compression which is based on the PBPM algorithm [10]. PBPM is a highly efficient compression algorithm that can dynamically learn frequent patterns in the data. To avoid the complexity of data location, pair-matching is proposed so that only two cache lines can be compressed together and stored in a single cache block. Though our work is based on FPC, it can also be easily extended to use other algorithms such as the PBPM algorithm.

Besides cache compression, memory compression is also a well studied topic. IBM's Memory Expansion Technology (MXT) [53] was commercially implemented in the Pinnacle [54] chip . MXT has a hardware compression engine built into the memory controller system. An off-chip uncompressed L3 cache is used to reduce the decompression latency. Sector Translation Table (STT) is used to translate a "real" address to a physical address. Every STT entry is mapped to a 1KB real address data block. If the line is compressed to be less than 120 bits, it is stored in the STT entry itself. Otherwise, the STT contains pointers that point to sectors of memory space. MXT uses Parallel Block-Referential Compression [14] algorithm to compress 1KB data blocks. The parallel implementation allows four 256 bit data blocks to be compressed at the same time, achieving a 4B/cycle compression rate. Because memory access tends to have relatively longer latency, it is not very sensitive to the compression latency. In addition, larger data blocks usually result better compression rate. In contrast, the cache is less resilient to the compression latency. Cache compression typically uses a lower latency compression algorithm that works on a smaller data block (usually 64B cache line). Hallnor and Reinhardt proposed a unified compressed memory hierarchy [21] that includes an Indirect-Index Cache Compression (IIC-C) and an MXT-like memory compression. Yang et al. studied an on-line memory compression scheme

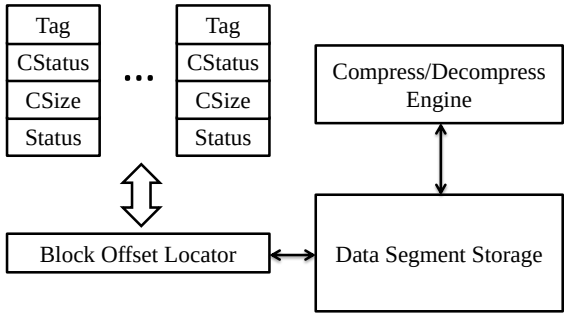

Figure 32: Decouple variable-segment cache diagram

for embedded system, called CRAMES [59]. CRAMES takes advantage of the operating system's virtual memory infrastructure. The swapped out memory pages could be stored in a compressed format. To swap in compressed memory pages, however, the program needs to wait longer due to the decompression. The "thread-aware" principle proposed in this chapter can potentially also be used in memory compression to selectively make compression decisions depending on whether the program is sensitive to the additional latency.

7.3 Motivation

In this section, we first briefly review a previous dynamic cache compression technique. We then show that conventional cache compression on multicores can lead to non-uniform performance speedups which motivate the work of this chapter.

7.3.1 Decoupled Variable-segment Cache

Alameldeen and Wood's dynamic cache compression technique is based on a decoupled variable-segment cache. To get the benefit of data compression, the cache must be able to hold more cache lines than the original number of lines. To achieve this, the number of cache tags first needs to be increased. For example, a 16-way cache can be augmented to have 32 tags per cache set, so one set can hold up to 32 compressed cache lines. The data storage per set still only corresponds to a 16-way cache, but the actual number of lines in the set can be greater than this depending on the compression rate. The cache data store is broken into eight-byte segments. One cache line can be in one segment or up to eight segments.

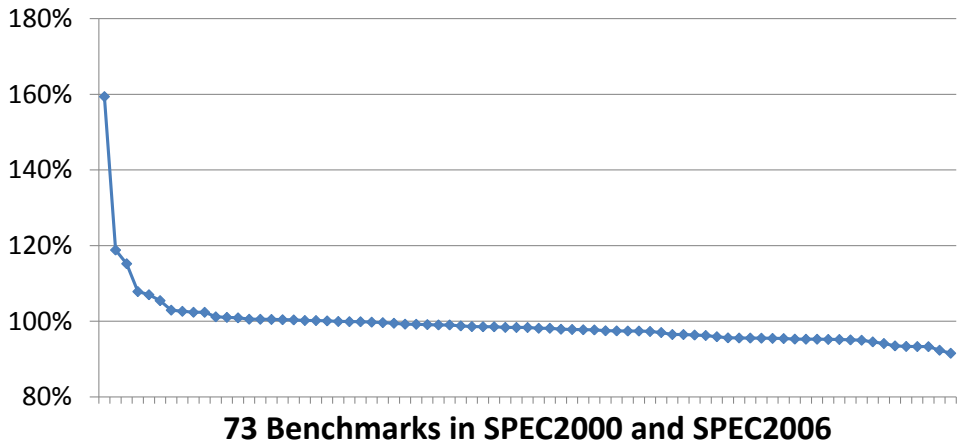

Figure 33: Single thread IPC speedup for a cache that always uses FPC compression, and a cache that never compresses. Every data point represents a benchmark with a particular input. The curve includes both SPEC 2000 and SPEC 2006 integer and floating point benchmarks

Because of the variable sized cache lines, the architecture needs mechanisms to know whether one line is in the cache and where the location of a given line's data is. For every cache tag, two fields are added: (1) CStatus, whether the cache line is in compressed form or regular form; (2) CSize, the cache line size in segments if the line is compressed. The cache lines are stored in contiguous data segments. Therefore, to locate the kth cache line, we need to compute the kth line's offset which is the sum of the CSizes of the first $k-1$ cache lines. A "Not-Present" status is also needed for the cache lines that are not able to fit in the data storage. Figure 32 shows a diagram of the decoupled variable segment cache structure.

The compression algorithm used in this chapter is Frequent Pattern Compression (FPC) which is a simple and effective compression algorithm [2]. It compresses a 64-byte line into one to eight 8-byte segments, according to different common patterns. For example, cache lines with all zeros or very small integers (such as 4-bit) can be compressed into one byte. A five-stage pipeline can be implemented to compress or decompress a cache line in five cycles. Their simulations reported a 1.0-2.4 compression rate for SPEC benchmarks and we also get similar results.

7.3.2 Non-uniform Applicability of Compression

The benefit of cache compression is that the effective cache size is larger so that the cache has fewer misses. The downside is that the cache access time is increased by the decompression latency. As discussed earlier, always compressing the cache sometimes improves the performance and sometimes it hurts. In the following experiment, we show the performance speedup for "always compress" compared to "never compress". Figure 33 shows the results for both SPEC 2000 and SPEC 2006 benchmarks, and they are sorted by the speedup. We can see that 14 benchmarks show positive speedup and most of the rest of the benchmarks show performance loses due to compression. This suggests that it is very important to make the correct compression decision, otherwise performance suffers.

Alameldeen and Wood's AdaptiveCC algorithm work can dynamically chooses whether to compress. They classify the cache accesses into four different types: (1) unpenalized hits, (2) penalized hits, (3) avoidable misses, (4) avoided misses and (5) unavoided misses. The access type can be determined by the cache hit position (LRU stack position) and whether there is a tag match. A penalized hit means a hit on a compressed line, and this hit would also be a hit if it was not compressed. A penalized hit indicates that compression is not helpful. On the other hand, the avoidable misses and the avoided misses indicate that the compression has helped or would help to reduce a cache miss. These two cases indicate that compression is indeed helpful. A global policy counter is incremented by the number of cycles of decompression latency or decremented by number of cycles of miss penalty for every cache access, depending on the classification of the access. The compression decision is determined by this counter value.

AdaptiveCC's global counter idea is able to capture the behavior of the current running thread. However, in a shared cache scenario it does not distinguish between different threads. The mechanism actually views the two or more concurrently running threads as a single big application and makes a global decision to enable/disable compression for the entire cache, and therefore for all applications. As Figure 33 shows how important the compression decision is to the performance, a wrong decision could result in up to 10% IPC slowdown or losing the opportunity to have up to 60% IPC speedup. In a shared cache, two

Table 10: Example Access Timer Tracker for a 8-way cache. The shaded blocks indicate cache lines fit in the 256-byte capacity.

Way No.	0 (MRU)	1	2	3	4	5	6	7 (LRU)
(TID, CSize)	(A,32)	(B,40)	(A,50)	(B,16)	(B,40)	(A,64)	(A,40)	(B,32)
$D_A = 0, D_B = 0$	64	64	64	64	64	64	64	64
$D_A = 1, D_B = 0$	32	64	48	64	64	64	40	64
$D_A = 0, D_B = 1$	64	40	64	16	40	64	64	32
$D_A = 1, D_B = 1$	32	40	50	16	40	64	40	32

threads' behaviors are mixed together so it can be difficult for AdaptiveCC to find a good compression decision. Suppose two benchmarks A and B have working set sizes of 100% and 30% of the entire cache size, respectively. AdaptiveCC is likely to enable compression because the sum of the working sets exceeds the cache size. However, if A can be compressed to 70%, a better decision is to only compress A but not B, so that both A and B fit in the cache while B does not have to experience unnecessary decompression latencies. As a result, we need an adaptive compression control mechanisms that treats threads differently, and this motivates our work "Thread-Aware Dynamic Cache Compression"(TADCC).

7.4 Thread-aware Dynamic Cache Compression

In this section, we will first introduce the Access Timer Tracker idea which estimates the performance impact of different compression decisions. To build a stable overall compression control system, we propose a Decision Switching Filter to carefully decide when to enable/disable decision switching on a per-core basis. Finally, we will show how TADCC interacts with cache management schemes and how they can together boost system performance.

7.4.1 Access Timer Tracker

To be able to use different compression decisions for different threads, we must measure the benefit of all of possible policy combinations. Take dual-core as example: for core A and core B, there are four possible decisions: (1) Neither compresses (decision=0), (2) A compresses but not B (decision=1), (3) B compresses but not A (decision=2) and (4) Both A and B compress (decision=3). When we load one data block into the cache, we compute

the CSize of that block, and store that with the tag. If the line is in the compressed form, it consumes CSize bytes in the data array of the cache. Otherwise the line is uncompressed and takes 64 bytes (line size). Independent of whether a line is currently compressed, we keep track of each line's CSize, and therefore we can compute the positions of all cache lines under different per-core compression decisions.

Table 10 shows an example of a cache set with 256 bytes of data storage. It therefore can hold four uncompressed 64-byte cache blocks or up to eight cache lines if they are compressed and fit (in theory, the data storage could hold even more than eight cache lines if they are highly compressible, but we are assuming only twice the number of tag entries which limits how many lines we can track). The first row of the table specifies the thread ID and the CSizes for each cache line. For example, (A,32) means the block is thread A's block and the CSize is 32 byte. The columns in the table are sorted by the LRU stack order from most recent (MRU) on the left to least recent (LRU) on the right.

For a given access to the cache, there are three possible access latencies. (1) A hit for a non-compressed line will take the normal cache access latency, say, 10 cycles. (2) A hit on a compressed line will take the normal cache access latency plus the decompression latency, say five cycles, for a total of 15 cycles. (3) A cache miss occurs if there is no matching tag (regular miss), or a matching tag exists, but the data does not fit in the cache (i.e., the first N cache lines do not compress tightly enough to allow this line to fit).

For a given access to this cache set, we can estimate the access latency by considering the matching tag location and CSizes of the lines that are more recent than the matched tag, and this can be performed for different per-core compression decisions. In the case of "Neither compresses" ($D_A = 0, D_B = 0$), all of the cache lines are uncompressed so they will each consume the full 64 bytes. Assuming the same 256-byte data storage as before, this means any tag match in the first four most recently used positions will result in a hit, and any older tags will result in a miss. In the case of "$D_A = 1, D_B = 0$", all of thread-A's lines are compressed and only consume a number of data bytes equal to their respective CSizes, while all thread-B's lines are 64 bytes. Suppose we have an access of B to way-4. In the case of "$D_A = 1, D_B = 0$", the cache would need 272 bytes to hold the five most

recent lines (ways 0-4) which exceeds the data capacity of 256 bytes, so this is a miss. In the case of "$D_A = 0, D_B = 1$", the cache needs only 224 bytes to hold the ways 0-4, and so an access to way-4 would result in a hit under this compression decision.

For every cache access, the Access Time Tracker (ATT) computes the access times for all four different compression decisions, by considering whether the access would have been an uncompressed hit, a compressed hit, or a miss. For any access sequence, the ATT tracks the cumulative access latencies for that sequence under each of the possible compression scenarios by accumulating the estimated access times in four counters. The latencies for compressed and uncompressed cache hits are constant. Memory access latencies can vary due to contention in the memory system, row buffer hits/misses, and other DRAM timing constraints. The ATT tracks the average memory access time and uses this to estimate the latency of a cache miss (contrast to the AdaptiveCC approach that simply uses a fixed constant memory access time).

After a fixed number of cache accesses (in our experiment it is set to 10000 accesses), we can compare the access times of the different compression decisions. We call this a decision era. The shortest access time likely indicates the best policy for the past decision era. The counters will be cleared and be ready for monitoring the next 10000 accesses. We can potentially use the best decision of the last decision era as a prediction for how to make the compression decision for the next era, but doing this alone leads to instabilities in the compression decisions that can degrade performance.

7.4.2 Decision Switching Filter

Though it seems natural to use the best decision according the last era, we found that the best compression decision sometimes switches too fast. Note that switching compression modes can introduce overhead. For example, when we switch from using compression to not using it, the cache data size is effectively expanding. The cache may not be able to hold the same number of blocks and one or more blocks may need to be evicted. If the victim blocks are dirty, we need to write them back. If the writeback buffer or the bus is not available, the cache must stall until the writebacks complete. (Lee et al. proposed Eager

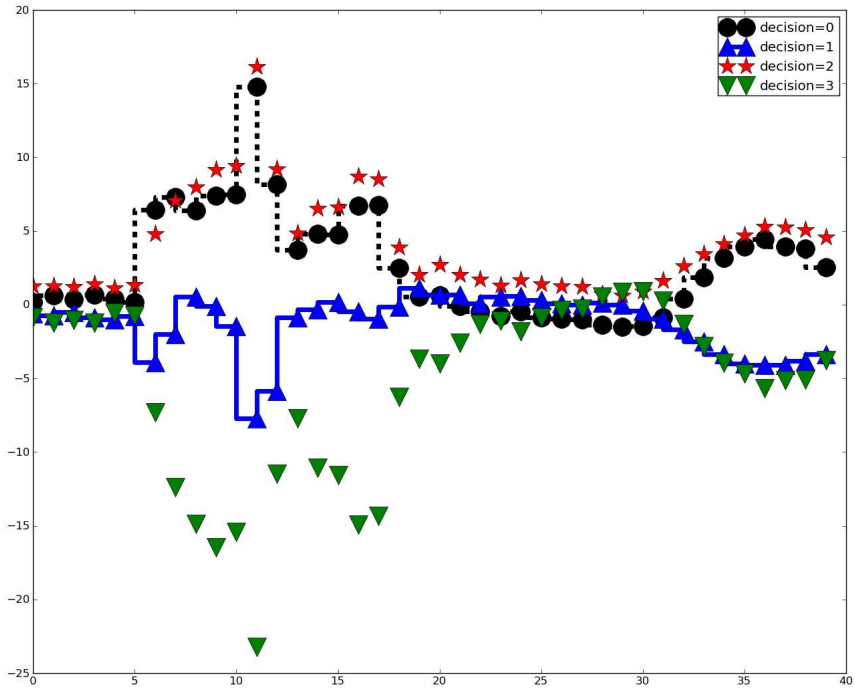

Figure 34: Access time measured by the Access Time Tracker. The lower the better. X-axis is time and Y-axis is the observed average access times for different decisions. They are normalized to the average of four decisions.

Writeback techniques which attempt to writeback dirty lines earlier than when they are to be evicted, depending on the current bus availabilty, and therefore the writeback traffic is redistributed and more balanced [34]). If the new best compression decision provides a significant expected performance improvement over the current decision (as indicated by a large difference in ATT counter values), then it may be worthwihile to pay the decision switching overhead. On the other hand, if the new best decision is only slightly better than the decision that is already being used, we would rather keep using the current compression decision and avoid the switching overhead. Figure 34 presents a snapshot of the observed access times (lower is better) for the workload of mcf and vpr-place for the four different policies. We can see that decision=3 ("$D_A = 1, D_B = 1$") provides a large advantage for a while, and then it becomes slightly worse than other decisions. Since the difference is small, however, we would rather keep using decision=3 than to pay the overhead of switching to another compression mode.

We need to filter out harmful decision changes when the expected benefit does not

outweigh the cost, but at the same time we want to quickly change when the performance improvement is significant. We propose a Decision Switching Filter (DSF) to achieve this. DSF has three parameters: (1) *low-threshold*, (2) *high-threshold* and (3) *decision-holdtime*. For every decision era, we check the difference between the average access times between the current decision and the new best decision as indicated by the ATT counters. If the difference is below the *low-threshold*, the new decision is ignored and the cache continues using the current compression decision. Sometimes the expected benefit is very large, and so if the difference is more than the *high-threshold*, we will immediately switch to the new decision. When the difference falls between the low-threshold and the high-threshold, this means that the difference is somewhat significant, but not large enough to justify taking the risk of switching to the new decision right away. If the new decision is consistently better for *decision-holdtime* consecutive eras, then we will switch to the new decision. We use the notation "*a*-*b*-*c*" to represent a configuration of DSF, where a, b and c are the three parameters discussed above. We found that it turns out "4-8-8" gives the best performance result (i.e., if the average latency improvement is less than four cycles per access, then do not change; if it is greater than eight cycles per access, then change immediately; if it is in between for eight consecutive eras, then change).

Note that when the cache changes compression decisions, the new decision is not immediately applied to the entire cache. The compression decision is only applied to newly inserted/modified cache lines. It could be either a write hit on an existing line or a cache fill from memory. One can potentially apply the compression decision to all the lines in the set that is being accessed. However this would introduce more overhead, and we did not observe much performance difference for doing this and so we do not further consider this in our experiments.

7.4.3 Interaction with Cache Management

The shared cache could have contention problems when multiple cores are competing for cache capacity. Shared cache management has been a well studied topic in the past several years [27, 28, 45]. While cache compression can help to increase the effective cache size to

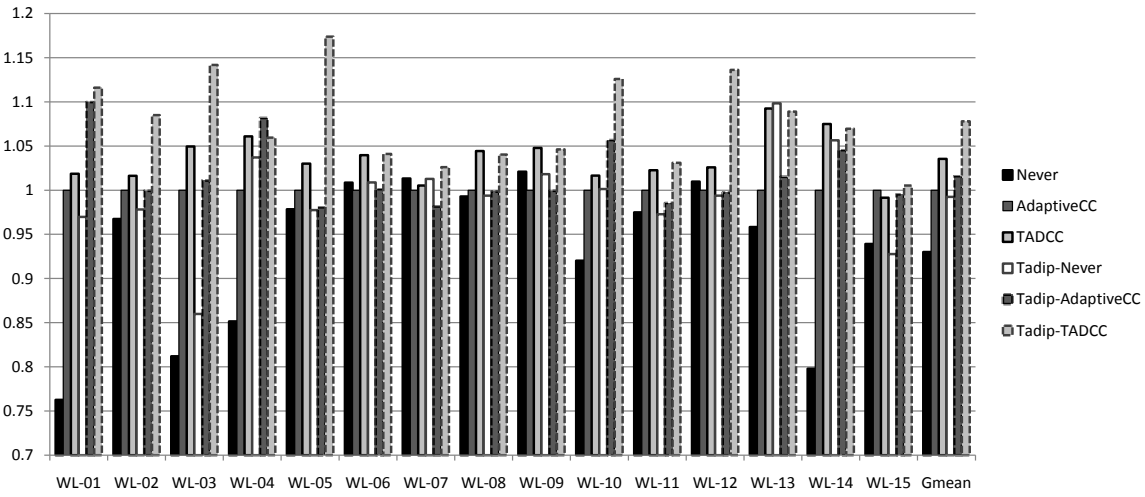

Figure 35: Performance comparision of TADCC and AdaptiveCC. All the performance number are IPCs normailzed to AdaptiveCC without TADIP.

help solve contention problems in some cases, we found that combining a cache management scheme with compression can even be better to further improve performance.

An example contentious scenario can occur when core A accesses the cache very frequently and has many cache lines that are *not* reused, while core B has a large foot print and needs a lot of cache space. TADIP [27] is an interesting cache management scheme that attempts to insert new incoming cache lines to the Least Recent Used (LRU) position instead of the traditional Most Recently Used (MRU) position. In this example scenario, TADIP would insert core B's cache lines to the MRU postition, and tend to insert core A's cache lines to LRU position so core A is prevented from evicting core B's useful data. If we do not employ a cache management scheme, core A may constantly evict core B's data. In this scenario, cache compression (whether thread-aware or not) becomes irrelevant, because whether compressed or not, core A does not get a chance to reuse its lines before core B thrashes the cache and causes core A's lines to be evicted. If a cache makes use of some sort of cache management scheme (such as TADIP), then the thread-aware dynamic compression scheme will have more opportunity to do a better job.

7.4.4 Hardware Cost and Reducing Power

Thread-Aware Dynamic Cache Compression (TADCC) has similar complexity as the AdaptiveCC approach. TADCC requires a small amount of additional combinatorial logic to

compute the hit positions and expected latencies for different compression decisions; these can be performed in parallel or sequentially (the latency is not on the cache's critical path). It also needs some simple MUXes for each tag to select between the actual CSize or the uncompressed cache line size. For a two core system, four counters are needed to store the access times.

Note that we do not need to update the counters very quickly, because we only need the values after a relatively long time interval. We can even sample accesses in time (e.g., only compute and accumulate the expected latency on every Nth access). In many cases, the best compression decision is stable for long periods of time, and so once a stable decision has been found, the detailed monitoring can be turned off for a while. One could rely on other simpler forms of phase detection (e.g., is there a sudden change in the cache miss rate?) to reactivate the ATT and DSF. TADCC relies on comparing the estimated access times for all possible compression decisions. For a two-core system, there are only four cases to consider. In a system with more than two cores, it might become expensive to evaluate the exponential space of possible decisions. Additional techniques such as set sampling, hill-climbing or other methods may be needed to modify TADCC to handle more cores.

7.5 Performance Evaluation

In this section, we first introduce the simulation framework we are using, and then we present the performance comparison of AdaptiveCC, TADCC and their interaction with cache management scheme TADIP.

7.5.1 Simulation Enviroment

This work is evaluated under the cycle-level model Zesto [36] which is an x86 simulator based on SimpleScalar [4]. The processor cores are loosely based on the AMD Bobcat processor [7]. The decoder can decode and issue up to 2 uops per cycle. We used a 56-entry ROB, 36-entry RS, 26-entry LDQ and 22-entry STQ. The processor has two cores. Each core has a private 32KB, 8-way L1 data cache and a private 32KB, 2-way L1 instruction cache. The last level cache (LLC) is 2MB, 16-way, which is shared between the two cores. All cache lines are 64 bytes (when not compressed). The memory sub-system uses a SDRAM

Table 11: Two-core workloads evaluated

Workload No	Benchmark 1	Benchmark 2
WL-01	mcf	vpr-place 2
WL-02	art-470	perl-diffmail
WL-03	art-470	twolf
WL-04	soplex-pds	vpr-place
WL-05	soplex-pds	gzip-soruce
WL-06	mcf	crafty 2
WL-07	mcf	gzip-program
WL-08	omnetpp	gcc-expr
WL-09	omnetpp	perl-diffmail
WL-10	gzip-source	mcf
WL-11	crafty	soplex-pds
WL-12	gzip-program	art-470
WL-13	vpr-place	omnetpp
WL-14	twolf	soplex-pds
WL-15	gcc-expr	galgel

model that is operating with 9-9-9 timing on a 800MHz front-side bus (effective 1.6GHz with DDR2).

Table 11 shows the two-core workloads evaluated in this work. These workloads are created mainly by combining a compression-friendly application and another one that is not. Performance is reported as the IPC throughput (IPC sum of both cores).

7.5.2 Performance Results

In this section, we will compare the TADCC with AdaptiveCC in two cases: under an unmanaged cache and under a cache that is running the TADIP management scheme.

In Figure 35, the IPC sum number is normalized to AdaptiveCC. With an unmanaged the cache, TADCC performs on average 4% better than AdaptiveCC. When TADIP is enabled for both, TADCC performs 8% better than AdaptiveCC. These results show how TADIP can increase the opportunity for TADCC to do better than AdaptiveCC.

There are workloads for which the AdaptiveCC is actually performing worse than the "Never" policy, such as WL-09. This is because the mixed behavior from two applications confuses the AdaptiveCC policy. A trace of the compression decisions shows that AdaptiveCC is choosing to enable compression almost all of the time. In this case, the

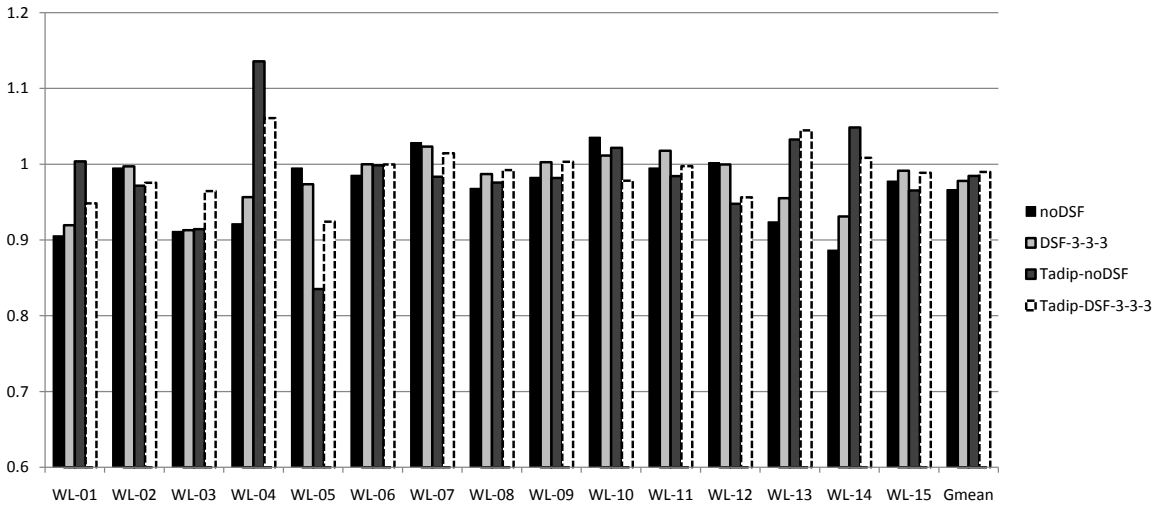

Figure 36: Performance slowdown if the Decision Switch Filer's parameters are set to something other than 4-8-8.

TADCC is still able to perform better, because the TADCC approach can selectively enable compression for each individual core, while AdaptiveCC uses a single counter to get an approximation of the benefit from compressing the entire cache.

There are cases where TADCC performing very well, such as WL-01, WL-03 and WL-05. We found that in these cases, the correct decision tends to be that one core needs compression while the other does not. Figure 37 shows one snapshot of the results observed by the Access Time Tracker. We can see that "decision=2" ($D_A = 0$ and $D_B = 1$) keeps being the best decision for a long time. The ATT would capture this behavior and make the best choice. In contrast, the AdaptiveCC will only be able to choose between decision=0 or decision=3, which are not the best choices.

The Decision Switching Filter is very important for the TADCC. In the previous results, the three parameters are set to 4-8-8. In Figure 36, we show the performance results (normalized to using 4-8-8) if there is no DSF or if the parameters are set to other values. In the case of an unmanaged cache (no TADIP), removing the DSF degrades performance by 4% on average and up to more than 10%. When TADIP is enabled, the average performance loss of removing the DSF is less, but the variance is much greater with a worst-case observed performance loss of 16% on WL-05.

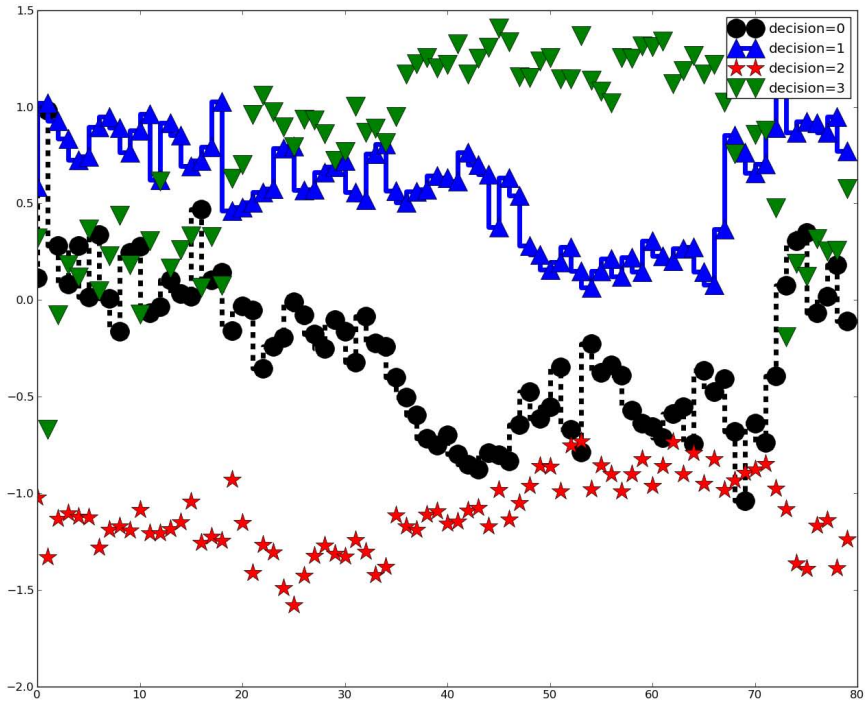

Figure 37: Access time comparison for different compression decisions. The X-axis is time and the Y-axis is the observed average access time normalized to the average of four access times. The lower the better. The data point is measured for every 10000 cache accesses. The lower the better.

7.6 Conclusion

Cache compression is a potentially useful approach to increase the effective cache size to reduce cache misses. Previous studies considered only single-core scenarios, and those proposed mechanisms can get confused by the variety of memory access behaviors present with multiple cores. In this chapter, we propose Thread Aware Dynamic Cache Compression (TADCC) which is able to make different decisions for different cores. The Access Time Tracker (ATT) dynamically tracks the average access time for all possible compression decisions, and the Decision Switching Filter (DSF) provides stability that can greatly reduce the harmful effects due to unnecessary fluctuations in the compression decisions.

This chapter investigates the interaction between a shared cache management scheme and cache compression. We found that first employing TADIP can help the cache to have a more balanced occupancy between cores, which then further gives TADCC more opportunities to do a better job on compression decisions. As a result, combining TADCC and

TADIP gives us 8% performance speedup on average and up to 17% compared to AdaptiveCC. This is an important result, as it shows that the full benefits of cache compression can be obscured by a poorly-behaving (unmanaged) cache.

There exist multiple directions for future work in thread-aware cache compression. For example, we can evaluate TADCC with other compression algorithms such as PBPM. To make TADCC scalable to more than two cores, we also need to study better algorithms to avoid evaluating all the decision possibilities.

CHAPTER VIII

CONCLUSION

Shared resource management is a general problem in computer architecture design. This dissertation thoroughly investigates the hardware mechanisms to efficiently manage the shared last level cache in multicore processors.

To study the cache contention problem, one needs to first understand when and how the problem happens. This dissertation proposes an animalistic classification technique which can dynamically classify applications into four categories: Turtle, Sheep, Rabbit and Devil. The cache competition happens when "Rabbit" and "Devil" are running together, because the rabbit needs large amount of the cache space and the bad behavior devil hurts it. The evaluation shows that the animalistic classification can accurately predict when the cache competition happens and that is the situation where the cache management is needed.

Qureshi and Patt's Utility based Cache Partition technique first uses utility monitor to observe different core's utility and then partitions the cache according to the partition decision based on the marginal hit counts reported by the utility monitor. When we have more than two cores and larger associativity, an accurate and scalable cache partition algorithm is needed to quickly compute the optimal partition. For a c core system with w way associative cache, there are $O(w^c)$ different partition possibilities. The dynamic programming technique proposed in this dissertation can compute the optimal partition in $O(c \cdot w^2)$ steps. This technique makes the cache partitioning scalable to more cores and larger cache associativity.

Promotion/Insertion Pseudo Partitioning(PIPP) proposed in this dissertation presents new ideas for cache replacement algorithms. In this technique, the newly inserted lines do not go to MRU position nor LRU position. The insertion position is dependent on the current cache allocation of that core. PIPP also does differently on a cache hit: it promotes the requested line to MRU direction by only one step instead of directly promoting it to

the MRU position. By doing different insertions and promotions, PIPP is able to combine the benefits of both cache capacity management and the dead cache lines management so that it outperforms the previous proposals UCP and TADIP.

As suggested by the animalistic classification, the cache competition problem is mainly caused by the bad behavior "devil". The Thrasher Caging technique in this dissertation explicitly controls the devil, preventing it from hurting other applications. This simple and effective technique needs less storage overhead and also does not need complex partitioning logic. As a result, Thrasher Caging is scalable to more cores and larger caches.

Most modern processors choose to use pseudo LRU algorithm for the last level cache. This dissertation studies how to approximate the hit position inside the LRU stack. By using this information, we can design a utility based cache partitioning algorithm for pseudo LRU caches. It turns out that the approximation provides enough accuracy for the partitioning purpose.

The above techniques are all trying to manage a given cache space between all the sharers. The last work in this dissertation deals with this problem in a different way by compressing the cache, because compression can increase the effective cache size. One related work proposed a dynamic technique to switch between compress or not-compress because sometimes decompression latency results in performance slowdown. However it does not distinguish different threads. Our thread-aware dynamic cache compression technique can treat threads differently according to their behaviors. The evaluation shows a significant speedup over the previous technique.

In summary, as this dissertation demonstrates, the shared cache in multicore processors can have competition problems that can be predicted by the animal classification technique. PIPP and the Thrasher Caging techniques are both low overhead and effective cache management schemes. Dynamic programming and the pLRU cache partitioning are great improvements on previous proposed cache management schemes. The thread-aware dynamic cache compression technique manages the shared cache from a new prospective by combining both thread-aware compression and management together. **We conclude that the shared caches in multicore processors need to be and can be managed to**

improve overall system performance by adding low overhead hardware.

This dissertation also motivates interesting future works. Industry becomes interested in integrating CPUs and GPU core on the same chip. Intel's new Sandy Bridge has a shared L3 cache between CPU and GPUs. Under this new architecture, the traditional cache management algorithm might not work well and therefore it gives opportunities to study new cache management schemes. The cache management is also interacting with other share resources such as the off-chip memory bandwidth. It will be interesting to study the relationship between the cache management/compression and the memory bandwidth. Power management is another important aspect of architecture design. Recently researchers are interested in studying how to dynamically distribute power budget between cores on the same chip. Potentially we can try to apply methodologies in this dissertation to the power partition problem, such as the animalistic classification and thrasher caging.

REFERENCES

[1] Alameldeen, A. R. and Wood, D. A., "Adaptive cache compression for high-performance processors," in *Proceedings of the 31st annual international symposium on Computer architecture*, ISCA '04, (Washington, DC, USA), IEEE Computer Society, 2004.

[2] Alameldeen, A. R. and Wood, D. A., "Frequent pattern compression: A significance-based compression scheme for l2 caches," tech. rep., University of Wisconsin-Madison, 2004.

[3] Alexander, G., Frommer, S., Levitan, D., and Sinharoy, B., "Thread-specific branch prediction by logically splitting branch history tables and predicted target address cache in a simultaneous multithreading processing environment," Oct. 10 2006. US Patent 7,120,784.

[4] Austin, T., Larson, E., and Ernst, D., "Simplescalar: An infrastructure for computer system modeling," *Computer*, vol. 35, pp. 59–67, February 2002.

[5] Bader, D. A., Li, Y., Li, T., and Sachdeva, V., "Bioperf: A benchmark suite to evaluate high-performance computer architecture on bioinformatics applications," in *In Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, p. 2005.

[6] Behar, M., Mendelson, A., and Kolodny, A., "Trace cache sampling filter," vol. 25, (New York, NY, USA), ACM, February 2007.

[7] Burgess, B., Cohen, B., Denman, M., Dundas, J., Kaplan, D., and Rupley, J., "Bobcat: AMD's Low-Power x86 Processor," in *IEEE Micro, vol.31, no.2, March/April 2011.*

[8] Chandra, D., Guo, F., Kim, S., and Solihin, Y., "Predicting inter-thread cache contention on a chip multi-processor architecture," 2005.

[9] Chang, J. and Sohi, G. S., "Cooperative cache partitioning for chip multiprocessors," in *Proceedings of the 21st annual international conference on Supercomputing*, ICS '07, (New York, NY, USA), pp. 242–252, ACM, 2007.

[10] Chen, X., Yang, L., Dick, R., Shang, L., and Lekatsas, H., "C-pack: a high-performance microprocessor cache compression algorithm," vol. 18, pp. 1196–1208, IEEE, 2010.

[11] Chiou, D., *Extending the Reach of Microprocessors: Column and Curious Caching.* PhD thesis, Massachusettts Institute of Technology, 1999.

[12] Choi, S. and Yeung, D., "Learning-based smt processor resource distribution via hill-climbing," 2006.

[13] Doweck, J., "Inside Intel Core Microarchitecture and Smart Memory Access," white paper, Intel Corporation, 2006. http://download.intel.com/technology/architecture/sma.pdf.

[14] Franaszek, P., Robinson, J., and Thomas, J., "Parallel compression with cooperative dictionary construction," in *Data Compression Conference, 1996. DCC'96. Proceedings*, pp. 200–209, IEEE, 1996.

[15] Fritts, J., Steiling, F., and Tucek, J., "MediaBench II video: expediting the next generation of video systems research," vol. 5683, p. 79, 2005.

[16] Garey, M. R. and Johnson, D. S., *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[17] Ghasemzadeh, H., Sepideh Mazrouee, S., and Kakoee, M. R., "Modified pseudo lru replacement algorithm," in *Proceedings of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems*, (Washington, DC, USA), pp. 368–376, IEEE Computer Society, 2006.

[18] Guo, F., Solihin, Y., Zhao, L., and Iyer, R., "A Framework for Providing Quality of Service in Chip Multi-Processors," in *A Framework for Providing Quality of Service in Chip Multi-Processors*, 2007.

[19] Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., and Brown, R. B., "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, (Washington, DC, USA), pp. 3–14, IEEE Computer Society, 2001.

[20] Hallnor, E. and Reinhardt, S., "A fully associative software-managed cache design," in *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, pp. 107–116, IEEE, 2000.

[21] Hallnor, E. and Reinhardt, S., "A unified compressed memory hierarchy," 2005.

[22] Hallnor, E. G. and Reinhardt, S. K., "A compressed memory hierarchy using an indirect index cache," in *Proceedings of Workshop Memory Performance Issues, 2004, pp. 9-15.*

[23] Hamerly, G., Perelman, E., Lau, J., and Calder, B., "Simpoint 3.0: Faster and more flexible program phase analysis," *Journal of Instruction Level Parallelism*, vol. 7, no. 4, 2005.

[24] Ipek, E., Mutlu, O., and others, "Self-optimizing memory controllers: A reinforcement learning approach," in *International Symposium on Computer Architecture*, pp. 39–50, IEEE, 2008.

[25] Iyer, R., "Cqos: a framework for enabling qos in shared caches of cmp platforms," in *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, (New York, NY, USA), pp. 257–266, ACM, 2004.

[26] Iyer, R., Zhao, L., Guo, F., Illikkal, R., Makineni, S., Newell, D., Solihin, Y., Hsu, L., and Reinhardt, S., "Qos policies and architecture for cache/memory in cmp platforms," in *Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '07, (New York, NY, USA), pp. 25–36, ACM, 2007.

[27] Jaleel, A., Hasenplaugh, W., Qureshi, M., Sebot, J., Steely Jr, S., and Emer, J., "Adaptive insertion policies for managing shared caches," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pp. 208–219, ACM, 2008.

[28] Jaleel, A., Theobald, K., Steely Jr, S., and Emer, J., "High performance cache replacement using re-reference interval prediction (RRIP)," in *Proceedings of the 37th annual international symposium on Computer architecture*, pp. 60–71, ACM, 2010.

[29] Kharbutli, M. and Solihin, Y., "Counter-based cache replacement algorithms," in *Proceedings of the 2005 International Conference on Computer Design*, ICCD '05, (Washington, DC, USA), pp. 61–68, IEEE Computer Society, 2005.

[30] Kim, S., Chandra, D., and Solihin, Y., "Fair cache sharing and partitioning in a chip multiprocessor architecture," in *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, p. 122, IEEE Computer Society, 2004.

[31] Kim, S., Chandra, D., and Solihin, Y., "Fair cache sharing and partitioning in a chip multiprocessor architecture," in *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, (Washington, DC, USA), pp. 111–122, IEEE Computer Society, 2004.

[32] Kron, J. D., Prumo, B., and Loh, G. H., "Double-DIP: Augmenting DIP with Adaptive Promotion Policies to Manage Shared L2 Caches," in *Proceedings of 2nd Workshop on Chip Multiprocessors Memory Systems and Interconnects (CMP-MSI-2)*, 6 2008.

[33] Lee, C., Potkonjak, M., and Mangione-Smith, W. H., "Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems," in *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 30, (Washington, DC, USA), pp. 330–335, IEEE Computer Society, 1997.

[34] Lee, H., Tyson, G., and Farrens, M., "Eager writeback-a technique for improving bandwidth utilization," in *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pp. 11–21, ACM, 2000.

[35] Lin, J., Lu, Q., Ding, X., Zhang, Z., Zhang, X., and Sadayappan, P., "Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems," in *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, pp. 367–378, IEEE, 2008.

[36] Loh, G. H., Subramaniam, S., and Xie, Y., "Zesto: A cycle-level simulator for highly detailed microarchitecture exploration," in *In Proc. of the Int. Symp. on Performance Analysis of Systems and Software*, 2009.

[37] Luo, K., Gummaraju, J., and Franklin, M., "Balancing thoughput and fairness in SMT processors," in *Performance Analysis of Systems and Software, 2001. ISPASS. 2001 IEEE International Symposium on*, pp. 164–171, IEEE, 2001.

[38] Mattson, R. L., Gecsei, J., Slutz, D. R., and Traiger, I. L., "Evaluation Techniques for Storage Hierarchies," *IBM J. Research and Development*, vol. 9, no. 2, pp. 78–117, 1970.

[39] Moore, G. and others, "Cramming more components onto integrated circuits," *Proceedings of the IEEE*, vol. 86, no. 1, pp. 82–85, 1998.

[40] Moreto, M., Cazorla, F., Ramirez, A., and Valero, M., "Explaining dynamic cache partitioning speed ups," *Computer Architecture Letters*, vol. 6, no. 1, pp. 1–4, 2007.

[41] Narayanan, R., zs. Ikylmaz, B., Zambreno, J., Memik, G., and Choudhary, A., "Minebench: A benchmark suite for data mining workloads," in *2006 IEEE International Symposium on Workload Characterization*, pp. 182–188, 2006.

[42] Nesbit, K., Aggarwal, N., Laudon, J., and Smith, J., "Fair queuing memory systems," in *39th Annual IEEE/ACM International Symposium on Microarchitecture, 2006. MICRO-39*, pp. 208–222, 2006.

[43] Olukotun, K., Nayfeh, B., Hammond, L., Wilson, K., and Chang, K., "The case for a single-chip multiprocessor," in *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pp. 2–11, ACM, 1996.

[44] Qureshi, M., Jaleel, A., Patt, Y., Steely, S., and Emer, J., "Adaptive insertion policies for high performance caching," in *Proceedings of the 34th annual international symposium on Computer architecture*, p. 391, ACM, 2007.

[45] Qureshi, M. and Patt, Y., "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 423–432, IEEE Computer Society, 2006.

[46] Rafique, N., Lim, W.-T., and Thottethodi, M., "Architectural support for operating system-driven cmp cache management," in *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, PACT '06, (New York, NY, USA), pp. 2–12, ACM, 2006.

[47] Ramsay, M., Feucht, C., and Lipasti, M., "Exploring efficient SMT branch predictor design," in *Workshop on Complexity-Effective Design, in conjunction with ISCA*, vol. 26, Citeseer, 2003.

[48] Rixner, S., Dally, W., Kapasi, U., Mattson, P., and Owens, J., "Memory access scheduling," in *Proceedings of the 27th International Symposium on Computer Architecture, 2000*, pp. 128–138, 2000.

[49] S., B. D., Strout, M., and Beveridge, J. R., "Faceperf: Benchmarks for face recognition algorithms," in *Proceedings of the 2007 IEEE 10th International Symposium on Workload Characterization*, IISWC '07, (Washington, DC, USA), pp. 114–119, IEEE Computer Society, 2007.

[50] Srikantaiah, S., Kandemir, M., and Irwin, M. J., "Adaptive set pinning: managing shared caches in chip multiprocessors," in *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIII, (New York, NY, USA), pp. 135–144, ACM, 2008.

[51] Suh, G. E., Rudolph, L., and Devadas, S., "Dynamic Partitioning of Shared Cache Memory," *JOURNAL OF SUPERCOMPUTING*, vol. 28, no. 1, pp. 7–26, 2004.

[52] Suh, G., Rudolph, L., and Devadas, S., "Dynamic partitioning of shared cache memory," *The Journal of Supercomputing*, vol. 28, no. 1, pp. 7–26, 2004.

[53] Tremaine, R., Franaszek, P., Robinson, J., Schulz, C., Smith, T., Wazlowski, M., and Bland, P., "IBM memory expansion technology (MXT)," *IBM Journal of Research and Development*, vol. 45, no. 2, pp. 271–285, 2001.

[54] Tremaine, R., Smith, T., Wazlowski, M., Har, D., Mak, K., and Arramreddy, S., "Pinnacle: IBM MXT in a memory controller chip," *Micro, IEEE*, vol. 21, no. 2, pp. 56–68, 2001.

[55] Tullsen, D., Eggers, S., Emer, J., Levy, H., Lo, J., and Stamm, R., "Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor," *ACM SIGARCH Computer Architecture News*, vol. 24, no. 2, pp. 191–202, 1996.

[56] Tullsen, D., Eggers, S., and Levy, H., "Simultaneous multithreading: Maximizing on-chip parallelism," in *ACM SIGARCH Computer Architecture News*, vol. 23, pp. 392–403, ACM, 1995.

[57] Xie, Y. and Loh, G., "Dynamic classification of program memory behaviors in CMPs," in *Proceedings of 2nd Workshop on Chip Multiprocessors Memory Systems and Interconnects (CMP-MSI-2)*, June 2008.

[58] Xie, Y. and Loh, G., "PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches," in *ACM SIGARCH Computer Architecture News*, vol. 37, pp. 174–183, ACM, 2009.

[59] Yang, L., Lekatsas, H., and Dick, R. P., "High-performance operating system controlled memory compression," in *Proceedings of Design Automation Conference 2006*.

[60] Yeh, T. Y., Faloutsos, P., Patel, S. J., and Reinman, G., "Parallax: an architecture for real-time physics," in *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, (New York, NY, USA), pp. 232–243, ACM, 2007.